PEARSON

·新·锐·编·程·语·言·集·萃·

The D Programming Language

D程序设计语言

【美】Andrei Alexandrescu 著

张雪平 谭丽娜 译



人民邮电出版社
POSTS & TELECOM PRESS

·新·锐

The D Programming Language

D程序设计语言

【美】Andrei Alexandrescu 著

张雪平　谭丽娜 译



北航　　C1645675

人 民 邮 电 出 版 社

北　京

图书在版编目（CIP）数据

D程序设计语言 / (美) 亚历山德雷斯库(Alexandrescu,A.) 著 ; 张雪平, 谭丽娜译. -- 北京 : 人民邮电出版社, 2013.6
（新锐编程语言集萃）
书名原文: The D programming language
ISBN 978-7-115-31419-2

Ⅰ. ①D… Ⅱ. ①亚… ②张… ③谭… Ⅲ. ①程序语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第061783号

内 容 提 要

D 语言是一种通用的系统和应用编程语言，它保持了生成高效代码以及直接访问操作系统 API 和硬件的能力。这是一本介绍 D 语言的权威书籍，全面、系统、专业地讲解了 D 语言相关的内容。书中涵盖了 D 语言的方方面面，不但介绍了语言的表达式、语句、类型、函数、契约、模块、类和结构等基本内容，还通过常见示例，深入解释了所有语言特性，讲解了如何将语言特性进行组合以支持重要的编程范型，阐述了每一项主要特性的基本原理，对它们的最佳用途给出了建议，并对跨领域的问题（如错误处理、契约编程和并发性）进行了讨论。此外，书中还通过大量的图表，为使用 D 语言解决日常问题提供了快速参考。其细致入微的讲解让人几乎觉得有些“啰嗦”，但正是这些“啰嗦”内容更能加深人们对 D 语言程序设计的理解，进而拓展到对一般意义上的编程语言程序设计的理解。

本书不仅适合于 D 语言初学者，对于想学习编译器实现的研究人员、想拓展学生的编程语言知识面的教师以及想深入了解编程语言设计的人，也能提供很好的帮助和指导。

◆ 著　[美] Andrei Alexandrescu
译　张雪平　谭丽娜
责任编辑　杨海玲
责任印制　程彦红　杨林杰
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷
◆ 开本：800×1000　1/16
印张：22
字数：493 千字　2013 年 6 月第 1 版
印数：1–2 500 册　2013 年 6 月北京第 1 次印刷
著作权合同登记号　图字：01-2012-6497 号

定价：69.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

译者序

没错，本书介绍的是D语言——这是一种集成了多种优秀特性（如内置字符串支持、基于垃圾回收机制的内存自动管理模式、自动类型推断等）且拥有系统级编程能力的语言。它支持多种编程范型、可以实现跨平台编程，同时无需依赖任何虚拟机。

尽管我一直都知道C/C++语言在软件开发领域有着广泛的影响力和优秀的表现，但对其手动管理内存的模式与编译速度一直耿耿于怀，且一直惊叹于Pascal语言的编译速度。直到2006年前后，我开始接触D语言，便立即被其优美的语法和一闪而过的编译速度所吸引。当时，D语言仍未发布稳定版本，各种形式的参考资料都比较少，比较权威的还是包含在DMD（D语言参考编译器）发行版里的参考文档。于是我试着通过翻译这些参考文档来加深对D语言的了解和学习，同时也希望这些翻译的资料能为国内更多D语言爱好者和追求高效编程的朋友提供帮助。随着D语言1.0稳定版本的发布以及后来D语言2.0的出现，这些参考文档的翻译也尽可能保持了同步更新。到2011年，我基本完成了把D语言2.0版本里所有参考文档的中文翻译。

对广大D语言初学者来说，DMD里的参考文档具有重要的参考价值，但这些资料主要还是侧重于语法方面的讲述，而且必要的实例都比较精简，对缺乏一定编程基础的人来说，理解这些资料仍然需要花费很大的力气。本书的出现将弥补这一缺憾，它全面、系统、专业地讲解了D语言相关的内容，其细致入微的讲解让人几乎觉得有些“啰嗦”。但正是这些“啰嗦”的内容更能加深人们对D语言程序设计的理解，进而拓展到对一般意义上的编程语言程序设计的理解。

本书的英文版在发行之后，受到了众多D语言爱好者的追捧。希望本书中文版的翻译出版能激发大家学习D语言的热情，进一步推动D语言在国内的发展和应用。

本书由我和谭丽娜共同翻译完成。能参与本书中文版的翻译是我的荣幸，这里要特别感谢人民邮电出版社对国内D语言发展的关注，感谢信息技术分社杨海玲女士等人给予的无私帮助，也感谢所有为此付出努力的朋友们。限于译者能力，翻译内容如有不当之处，敬请大家批评指正，对可能造成的误解深表歉意！

张雪平

2013年2月于上海

自在读书
www.zizidiary.com

Walter Bright 序

记得很久以前，我读过一本科幻小说，其中有这么一行：科学家在可能拓展其研究领域知识的时候，就应该勇往直前。简单地讲，它揭示了想要成为一名科学家所应具备的潜质。值得一提的是，在物理科学家 Richard Feynman 的影像资料和著作中，无处不体现着发现的乐趣，而且他对科学的热情也极具感染力且非常迷人。

尽管我不是科学家，但我明白他们的动机。我的动机跟工程师的动机一样——享受创造的乐趣，从无到有把某样东西建造出来。有一本我很喜欢的书 *The Wright Brothers as Engineers*，作者是 Wald，书中记录了莱特兄弟在克服一个又一个飞行难题之后逐步前进的过程，同时还记录了他们如何应用这些知识来创建飞行器的过程。

我早期的兴趣与 Brinley 所著的 *Rocket Manual for Amateurs* 开篇中所概述的内容一样："燃烧与爆炸的事物总是让人激动和着迷"。后来又变成想去构建一些能跑得更快和飞得更高的事物。

不过构建功能强大的机器耗资巨大，于是我发现了计算机。它最神奇的地方在于可以让你轻松地去构建某些事物，又不需要昂贵的制造车间和机械工厂，甚至连一把螺丝刀也不需要。只需要一台花费不多的计算机，你就可以创造世界。

就这样，我开始在电脑里建造虚拟世界。第一个作品便是游戏 *Empire*——世纪之战。由于当时电脑的处理能力实在太弱，这个游戏根本就没办法正常运行，因此我又开始对如何优化程序的性能产生了兴趣。此兴趣让我开始对可以生成代码的编译器进行研究，自然而然便发展到了"我能写出比它更好的编译器"这样一种自负。因迷恋 C 语言，我迫切希望实现一个 C 语言编译器。这个也不是太难，大概只用了几年的空余时间。后来我又发现了 Bjarne Stroustrup 的 C++语言，于是我认为再花上几个月的时间应该能够将那些扩展添加到我的 C 语言编译器里。

十多年后，我仍然在为此努力。在实现这个编译器的过程中，我对 C++语言的每一个细节变得非常熟悉。为一个庞大的用户群提供支持意味着能体验到其他人对此种语言的大量的认知：起作用的是什么，而什么是没用的。我不会不假思索地去改进设计。在 1999 年，我决定将这一想法付诸实践。开始时它叫做 Mars 编程语言，而我的同事最初是开玩笑地称其为 D 语言，不过这个名称后来被延用下来，D 语言由此诞生。

在撰写本文时，D 语言已有十个年头，而且已演变到了第二代，有时也叫 D2。在这段时期里，D 语言已从由一个人敲打着键盘开发它，发展到由一个全球性的开发者社区来解决它方方面面的问题，以及为库和工具的生态系统提供支持。

D 语言自身（本书的焦点所在）已从卑微的初级阶段成长为一门功能非常强大的语言，它擅长从多种角度解决编程问题。据我所知，D 语言前所未有地将多种强大的编程范型

自在读书
www.zizidiary.com

（programming paradigm）巧妙地集成在一起：命令式（imperative）、面向对象（object-oriented）、函数式（functional）以及元式（meta）。

乍一看，这门语言肯定不简单。的确，D 语言不是一门简单的语言。但我认为这是一种错误地看待一门语言的方式。更恰当的方式应该是：这门语言的编程解决方案看起来的样子像什么？D 语言程序是复杂愚钝还是简洁优雅呢？

我有一位同事，他有着丰富的企业环境开发经验。他注意到 IDE（Integrated Development Environment，集成开发环境）是一个相当重要的编程工具，因为只需单击一下鼠标就会有数百行的样板代码生成。对 D 语言来说，IDE 不是不可或缺的，因为 D 语言并不依赖基于向导式的样板代码。通过自省（introspection）和生成功能，它让自己不再依赖于使用样板。程序员不必看到样板，负责程序内在复杂性的是语言，而非 IDE。

例如，有人想要用更简单一些的语言（对范型没有特别的支持）来进行 OOP（面向对象编程）设计。这也是完全可能的，但这种做法糟糕透顶，而且毫无价值。当有一门更为复杂的语言直接支持 OOP 时，编写 OOP 程序就会变得简洁优雅。语言越复杂，用户代码就越简单。这才是值得去做的事情。

必须有一种可以支持多范型的语言，才能支持以简洁优雅的方式为各种类型的任务编写用户代码。正确编写的代码在页面上看起来应该是赏心悦目的，说来也奇怪，漂亮的代码往往就是正确的代码。我不确定两者之间存在什么样的联系，但确实如此。与好看的飞机常常都飞得很好是一样的道理。因此，让算法可以以一种漂亮的方式来进行表达，这样的语言特性肯定会是件好事情。

然而，编写代码时的简洁优雅并不是衡量一门程序语言好坏的唯一标准。当前，程序的规模在无止境地迅速增长。随着这种变化，依赖于惯例和编程方面的专长以确保代码的正确性变得越来越不实际；相反，依赖于使用机器进行检查的保证正变得越来越有价值。为此，D 语言提供了多种策略，让程序员可以使用它们来实现这类保证。这些策略包括契约（contract）、内存安全（memory safety）、各种函数属性、不变性（immutability）、劫持保护（hijack protection）、作用域守护（scope guard）、纯洁性（purity）、单元测试（unit test）、线程数据隔离（thread data isolation）。

我们并没有忽视性能！尽管有预言说性能不再那么重要，尽管现在计算机的运行速度比我编写第一个编译器时快了上千倍，但对更快速程序的需求却从未改变。D 语言是一门系统编程语言。这预示着什么呢？从某种意义上讲，它意味着使用 D 语言除了能编写设备驱动程序和应用程序外，还可以用来编写操作系统。在更为技术的层面，它意味着 D 语言程序可以访问机器的所有功能，即可以使用指针、进行指针别名和指针运算、绕过类型系统，甚至可以使用汇编语言编写代码。没有什么是 D 语言程序员无法访问的。例如，D 语言的垃圾回收器实现就完全是使用 D 语言编写的。

请等等！这怎么可能？一门语言怎么能同时提供可靠的保证和随意的指针操作呢？答案就是这种保证是基于语言所使用的概念。例如，函数属性和类型构造函数可用于实施编译时的保证，而契约和不变量（invariant）则专门用于运行时实施的保证。

自在读书 www.zizidiary.com

D 语言的大部分特性都曾以这样或那样的形式出现在其他语言里。任何特定的单个特性都无法造就一门语言，但它们组合后所产生的作用却能超过单个部分所起作用的总和，D 语言的组合造就了一种令人满意的语言，它以优雅和直接的方式来解决各式各样的编程问题。

Andrei Alexandrescu 因其不拘一格的编程思想而闻名，这些思想已成为新的主流（请参考他的重要著作 *Modern C++ Design*）。Andrei 于 2006 年参与到 D 编程语言的设计团队。他拥有坚实的编程理论基础，以及无尽创新性的针对编程设计问题的解决方案。D2 的成型主要归功于他。本书在许多方面都跟随着 D 语言一起在演变。在这本关于 D 语言的著作中，你会欣喜地发现这样一件事件，即其中包含的是很多关于设计选择的原因，而非枯燥乏味的事实陈述。了解一门语言为什么会是这个样子能让人更容易和更快速地理解它，并让程序可以运行得更快。

使用 D 语言来解决大量基础性的编程问题，Andrei 不断地采用这种方式以求阐明其中的原由。因此，他不仅展示了 D 语言是如何工作的，而且还说明了它为什么会工作以及如何使用它。

正如我已把 D 语言融入我的生活一样，希望你也能在 D 语言编程的过程中获得乐趣。在 Andrei 的这本书里，每一页都令人兴奋不已，相信你一定会受益无穷。

Walter Bright
2010 年 1 月

自在读书
www.zizidiary.com

Scott Meyers 序

不管从哪个方面来讲，C++都获得了巨大的成功，但即使是其最狂热的拥护者也不可否认它是一只复杂难懂的“猛兽”。这种复杂性也影响到了被广泛使用的 C++后继者 Java 和 C#的设计。这两种语言都在尽量避免 C++的复杂性，采用的方法是，通过一个更易使用的程序包提供其大部分的功能。

降低复杂性可以采用的基本形式有两种。一种形式是，去掉“复杂的”语言特性。例如，通过垃圾回收（garbage collection）的方式可以避免 C++中对手动管理内存的需要。在模板的“成本/效益”测试被认定为失败时，这些语言的初始版本都选择了排除 C++中与泛型支持相关的所有内容。

降低复杂性的另一种形式是，使用相似但要求更少的结构方式取代“复杂的”C++功能。C++的多重继承可以改变成使用接口进行扩展的单继承。Java 和 C#的当前版本都支持模板化的泛型，但它们都比 C++的模板更简单。

这些后继语言并不只是要求通过降低复杂性来完成 C++所能完成的任务，而是被寄予了更多的期望。它们都定义了虚拟机，增加了对运行时反射（runtime reflection）的支持，并提供了大量的开发库，这些库可以让许多程序员将关注点从创建新代码转移到组合现有的组件方面上来。最终结果就是它们被看做是基于 C 语言的“生产力语言”（productivity language）。如果想快速创建软件，而它几乎等同于去组合现有的组件（很多软件都属于此类），那么与 C++语言相比，Java 和 C#都是不错的选择。

但 C++不是生产力语言，它是一种系统编程语言。它旨在媲美 C 语言在以下几个方面的能力：硬件通信（如驱动程序和嵌入式系统方面），在不经任何调整的情况下可以直接处理基于 C 语言的库和数据结构（如遗留系统方面），用尽运行硬件上的所有性能。这里不是想嘲讽 Java 和 C#语言底层虚拟机的关键性能组件都是使用 C++编写的，高性能虚拟机的实现就应该是系统语言干的事，不适合生产力语言。

D 语言的目标是想在系统编程领域成为 C++的继任者。像 Java 和 C#语言一样，D 语言也想要避免 C++的复杂性。为此，D 语言使用了一些相同的技术，其中包括加入垃圾回收功能，摒弃手动内存管理[①]；采用单继承和多接口，舍弃多继承。但是，D 语言走的是一条属于自己的路。

先是从标识出 C++的功能性漏洞并修补它们开始。当前的 C++未提供对 Unicode 的支持，而其后继版本（C++11）也只提供了有限的支持。D 语言从一开始就支持 Unicode。当前的 C++和 C++11 都不支持模块（module）、契约编程（contract programming）、单元测试（unit testing）或“安全”（safe）子集（其中不可能出现内存错误）。D 语言提供了上述所有功能，但未对生成高质量本地代码的能力产生影响。

针对 C++强大与复杂兼有的特点，D 语言瞄准的目标是：在保持功能强大的同时降低复杂

① 事实上，它是可选的。作为名符其实的系统编程语言，如果你真的想要手动管理内存，D 语言也不阻拦你。

自在读书
www.zizidiary.com

性。C++中的模板元编程人员已证明，编译时计算是一项很重要的技术，但必须要先克服句法的困难才能将它付诸实践。D 语言提供了类似的功能，但没有词法方面的烦恼。在当前的 C++里，即使知道如何编写函数，你也不一定了解如何才能编写出与之对应的能在编译期间进行计算的 C++函数。在 D 语言里，如果知道如何使用编写函数，你就能准确地知道如何编写其编译时版本，因为两种情形下的代码是一样的。

最有意思的地方是，在基于线程的并发性方面，D 语言与其由 C++派生出的兄弟语言之间使用的方法是截然不同的。D 语言认识到对共享数据不正确的同步访问（数据竞争）就是一个陷阱——容易陷进去且很难爬出来。因此，它从一开始就约定：默认情况下，数据不会被跨线程共享。正如 D 语言的设计者指出的那样，在现代硬件深度缓存层次的环境里，内存并不是真正跨内核或处理器共享的。既然这样，那为什么还要默认给开发者提供的一种抽象呢？这种抽象不仅仅是一种错觉那么简单，它是一种错误的观念——众所周知，是它滋生了难以调试的错误。

所有这些内容让 D 语言在 C 语言的传统设计空间里特别引人注意，这一点已足以成为你阅读这本书的理由。而本书的作者是 Andrei Alexandrescu，阅读理由变得愈加充分。作为 D 语言的共同设计者以及绝大部分 D 语言库的实现者，没人能比 Andrei 更了解 D 语言。他能对 D 编程语言进行描述，这是很自然的事，他也能解释 D 语言为什么是这个样子。有些特性出于某种原因存在于 D 语言里，而有些应该有的也正缺失的特性出于某种原因却未包含在 D 语言里。要阐明其中的原因，Andrei 是不可替代的人选。

这种解释独特而富有魅力。其中有些内容似乎有点跑题（但事实上它们只是通往终点的一个中间站），Andrei 对此作了保证："我知道你在反问自己：编译时计算真的有必要吗？很有必要！请容我慢慢道来。"针对连接器诊断不够直观的根本性问题，Andrei 说到"如果你忘记了`--main`，也不用着急。连接器会使用其本地语言——编码后的克林贡语[①]，流畅、详尽地[②]提醒你。"即使是对其他出版物的引用，Alexandrescu 也都有提及。他不是简单地像"Wadler 的著作 *Proofs are programs*"这个样子进行提及，实际提及的样式非常详实，如"Wadler 的迷人著作 *Proofs are programs*"；关于 Friedl 的著作 *Mastering regular expressions* 不是简单推荐，而是"热情推荐"。

一本关于编程语言的书自然满篇都会带有样例代码，而这些样例代码同时也说明 Andrei 是一个注重实践的作者。这里有一个他编写的搜索函数原型：

```
bool find(int[] haystack, int needle);
```

本书描述的是一门很有意思的编程语言，是由一位经验丰富的作者编写而成的。我保证你会发现这本书值得一读。

Scott Meyers
2010 年 1 月

① 英文原词 Klingon，它指的是一种人造语言，它主要的特点是使用"宾谓主结构"（OVS）语序，而这种语序在很多情形下正好与英文语序相反。——译者注

② 英文原词 baroquely，它是 baroque 一词的变形。baroque 是一个英语外来词，一般情形下它指代的是 17 世纪风行于欧洲的一种艺术风格——巴洛克。而在非正式场合，它的含义与 elaborate（详尽的）很接近。因此，这里将 baroquely 译作"详尽地"。——译者注

自在读书 www.zizidiary.com

前 言

设计编程语言追求的是简单而又不失功能的强大，在获得成功的同时也会享受其中的优美。

在各种相互矛盾的需求中权衡取舍是一项极具挑战性的任务——它要求语言的设计者尽可能精通各种理论原则，并且拥有实实在在的语言实现经验。设计编程语言其实就是一项彻头彻尾的软件工程。

D 语言是这样一种语言，它会坚定地朝着自己设定的目标发展，它会提供对计算资源系统级访问的能力，也会提供高性能的处理能力，以及保持与 C 语言的派生语言相类似的语法特性。在确保目标不变的情况下，D 语言有时会遵从传统，实现一些其他语言所拥有的功能；有时又会打破传统，提供全新的、创造性的解决方案。也就是说，D 语言会重新审视其表面声称的那些约束。例如，将大程序进行细分或将整个程序使用定义良好且内存安全的 D 语言子集来编写，尽管这样会损失少量进行系统级访问的能力，但却大大提高了程序的可调试能力。

如果下面列出的几点对你非常重要的话，那么你可能就会对 D 语言产生兴趣。

- 性能。D 语言是一门系统编程语言。它的内存模型尽管高度结构化，但却保持了与 C 语言的兼容性。它可以直接调用 C 语言的函数，也可以被 C 语言的函数调用，中间不需要进行任何转换。
- 表达力。D 语言不是小巧的简约型语言，它拥有很高的“推重比”[①]。使用 D 语言可以定义出具有说服力和清晰明了的设计，从而精确地对错综复杂的现实场景建立模型。
- “扭矩”。所有后院里的快车改装者都会告诉你，动力并不是一切，可用性才是最重要的。有些语言对于小程序来说是最强大的，还有些语言实践证明只适合于某种特定的规模，而 D 语言对简短的脚本和大型程序来说都很适合。如果一个大型程序能有序地从一个简单的单一文件脚本逐渐增长起来，这可是一件很了不起的事情。
- 并发性。与其他相似的语言相比，D 语言针对并发性的解决方法算是突破性的，就像现代的硬件设计与前两年的架构相比所产生的突破一样。D 语言破除了隐式内存共享的魔咒，能够创建几乎完全独立的线程（它们通过消息实现彼此间的通信）。
- 泛型代码。能够操控其他代码的泛型代码最早起始于功能强大的 Lisp 宏，后来被其他语言延续，其中包括 C++的模板（template）、Java 的泛型（generic）以及其他语言中类似的特性等。D 语言也提供了相当强大的泛型机制。
- 中庸之道。D 语言明白，不同编程范型在满足不同的设计需求方面是有益的，因此，它提供了高度集成的复合式编程风格来代替“仅限单选”的编程方式。
- “这是我的原则。如果不喜欢这些，我还有其他的。”D 语言会尽最大努力去发现语言设

① 这是一个航空专业术语，作者是想借用它来形容 D 语言表达力的强大。——译者注

自在读书 www.zizidiary.com

计中的固有原则。有时，这些原则也会遇到一些问题，如实现难度、适用性难度等。而其中首要的问题则是人的本性——盲目的一致性并不总是容易察觉，也并不总是直观的。在这种情况下，所有的语言都必须做出判断，而这种判断常常带有很强的主观性，与平衡性、灵活性以及好口碑等方面的内容也是息息相关。至少在我看来，D 语言非常客观地在参照其他已做过类似决定的语言。

读者对象

本书假定你是一名程序员，也就是说，你应该知道如何使用自己熟悉的语言来完成一些常规的编程任务。本书不会假定或者特别推荐你一定要具备某种特定语言的知识。但如果你熟悉的语言是由 Algol 发展来的（如 C、C++、Java 或者 C#），那么你可能会稍微有一点优势，因为从一开始，它们的语法是很相似的，甚至想找出一个不一样的语言（语法相似但语义不同）都很难。具体来讲，如果你粘贴一小段 C 语言代码到 D 语言文件里，它要么可以用相同的语义进行编译，要么根本就不用编译。

一本介绍语言的书，如果没有对各种特性背后的动因提供深入的剖析，也没有对使用这些特性来完成具体任务的最有效方式进行说明，那么它将是枯燥乏味的，同时也是不完整的。对于所有那些不太起眼的功能，本书也讨论了它们背后所蕴藏的机理；同时还对一些看起来很好的设计，最终却未被选中的原因进行了反复的说明。有些设计选择可能会大大加重语言实现的负担，也可能会对那些有着更好理由留在原地的特性置之不理，还可能会掩盖很多在简短示例里无法见到的责任，或者掩盖那些因不够强大而无法自我承载的责任。更为重要的是，语言设计者跟其他所有人一样常会犯错，因此可能有些好的设计已经存在，但就是未被发现。

本书组织

第 1 章概要地介绍 D 语言的各主要部分。这一章并没有涉及所有细节。不过你还是能从中建立起对 D 语言的初步印象，并使用它来编写一些小程序。第 2 章和第 3 章是不可或缺的语言参考章节，其中分别讲解了表达式和语句。我尝试着将它们与传统语言进行融合，在有差异的地方，则进行了突出显示。你可以依次阅读这些章节，也可以在需要的时候回头查阅。在这些章的末尾都备有一张“速查表”——用简洁、直观的词汇进行表达，以帮助快速记忆。

第 4 章讲述的是内建数组、关联数组以及字符串。数组可以理解为带有安全开关的指针，它是 D 语言实现内存安全的工具，也是大家享受 D 语言乐趣的工具。字符串是由经 UTF 编码后的 Unicode 字符构成的数组。对 Unicode 的支持遍及整个 D 语言和标准库，这为字符串处理的正确性和效率提供了保障。

在阅读前 4 章之后，你便可以使用 D 语言提供的抽象概念编写一些简短的脚本式程序。后续的章节介绍的是抽象概念的基础内容。第 5 章以整合的方式对函数进行了描述，其中包括编译时参数化函数（模板函数）以及编译期间求值的函数。这些概念在正常情况下应该被归到高级章节里，但 D 语言让它们变得简单了，完全可以提前介绍。

自在读书 www.zizidiary.com

第 6 章讨论基于类的面向对象设计。编译时参数化类再一次以一种高度集成、有机组合的方式出现在这里。第 7 章继续介绍一些类型，其中引人注目的是结构（struct）。结构是构建高效抽象概念的工具，它常与类（class）配合，一起使用。

接下来的 4 章讲述的是相对独立且有些特殊的功能。第 8 章涉及的是类型修饰符。修饰符能够提供很强的保证，这种保证在单线程和多线程应用程序里都能发挥作用。第 9 章谈及的是异常模型。第 10 章介绍 D 语言在“契约编程”方面的强大功能，并特意将它与第 9 章区分开来，其目的在于想要消除一种常见的误解：错误处理与契约编程实际上是一回事。其实不然，第 10 章会解释其中的原因。

第 11 章针对组件式大型程序的构建提供了部分信息和建议，并对 D 语言标准库做了简单梳理。第 12 章介绍的是操作符重载，如果没有它，许多像复数那样的抽象概念就会受到严重影响。最后，第 13 章讨论了 D 语言针对并发问题的原始解决方法。

发展简史

听起来有些俗气，但 D 语言就是一项充满爱的工作。Walter Bright 是一名 C 和 C++编译器作者。在 20 世纪 90 年代的某一天，他决定不再把继续维护编译器作为自己的事业。因此，在觉得“时机成熟”的时候，他便着手去定义一门新的语言。我们当中有很多人都梦想着在某个时候能去定义出讨人喜欢的语言。幸运的是，Walter 已经拥有了一大堆基础工具，其中包括后台代码生成器、连接器以及丰富的构建语言处理器的宝贵经验。尤其是那些经验，让 Walter 在设计时能够高瞻远瞩、洞悉全局。在某种神奇自然法则的作用下，语言功能设计不足的问题在复杂编译器的实现过程中便会暴露出来。在设计新语言时，Walter 也尝试过系统性地克服这类弊端。

在 D 语言的初期，它在思想上与 C++非常相似，尽管 Walter 最开始想将它命名为 Mars，但社区还是因此简单地称其为 D 语言。基于后面会提到的原因，我们已将这门语言称作为 D1。多年来，Walter 一直都在开发 D1。他靠着澎湃的激情和不懈的努力吸引了一群追随者。到 2006 年的时候，D1 已成长为一门强大的语言，在技术上已能跟 C++和 Java 那样的语言进行正面的竞争。不过，当时的形势已经很明朗：D1 不可能成为主流语言，因为它还没有足以令人神往的特性来弥补与其他语言的差距。于是，当时 Walter 做了一个大胆的决定：他认为 D1 是通往理想国度的第一个版本，应该把 D1 置于维护模式，然后去着手修订这门语言的第二轮设计，同时不再考虑向后兼容的问题。目前，D1 的用户能继续获得错漏[①]修复，但 D1 不会再加入新特性；而 D2 将变成最新的语言定义，以后我便把它称作 D 语言。

这个决定获得了成功。第一版的设计让大家明白哪些该做哪些需要避免。同时，也没有急着去对新语言进行宣传——新用户可以继续使用稳定且被积极维护的 D1。因此，兼容性和截止时间的压力都没有成为主要问题，同时也有了足够的时间进行仔细的分析和设计新的版本，并且可以完完全全地进行正确的决定。为进一步推动设计，Walter 还召集了像 Bartosz Milewski 和我这样的合伙人一起参与设计。在一次只有我们三人参加的、用时很长而且气氛活跃的聚会

① bug 原意为臭虫、错误、漏洞。本书统一将其译作“错漏”。——译者注

自在读书 www.zizidiary.com

上（地点就在华盛顿州柯克兰市的一家咖啡店里），许多与 D 语言直接相关的重要特性，如不变性（immutability）、泛型编程（generic programming）、并发性（concurrency）、函数式编程（functional programming）、安全性（safety）等，都被一一确定下来。

最终，D 语言没有辱没“更优 C++”的名号，并成为了一门强大的多用途语言，它可以同时承担系统级的、企业级的以及脚本语言的工作。这里仍然留有一个问题，那就是 D 语言的进步与革新主要是在背后进行的，因此关于使用 D 语言进行编程的方式并没有太多的文档进行记录。

大家正阅读的这本书就是想要填补这个空白。希望大家也像我喜欢写这本书一样，喜欢阅读它。

致谢

D 语言有许许多多的贡献者，在这里无法一一列举。其中有很多人在新闻组 digitalmars.D 里的表现都特别突出。新闻组就像是一块共鸣板，用于审视我们提出的设计，同时也激发了许多的想法和改进方法。

Walter 在社区也受益不少。在定义编译器 dmd 的参考实现方面，有两名贡献者的表现极为出众，他们是 Sean Kelly 和 Don Clugston。Sean 重新编写和改进了核心运行库（其中包括垃圾回收器），并且编写了大部分的 D 语言并发库实现代码。他很擅长自己的工作，如果你的并发代码出现问题，那么很可能是你自己的原因造成。Don 是一位数学领域的专家，尤其是在浮点数方面。他让 D 语言的数值原语（numeric primitive）成为最棒的实现之一，而且他还将 D 语言的更新换代能力提升到极限。在参考编译器 dmd 的源代码对外开放之时，Don 就开始不停地往里面添加代码，并成为了 dmd 的第二大贡献者。Sean 和 Don 两人都发起过改进 D 语言定义的提议并将其完成。另外还漏掉一点，他们都是非常优秀的“黑客”，很喜欢与人交流，无论是面对面的还是在线的，都包括在内。很难想象如果没有他们，D 语言会发展到何处去。

最后，我要向审读本书的人表示诚挚的谢意，感谢他们给予的理解。他们完成的是一项艰苦却又费力不讨好的工作。没有他们，本书就不会有现在的样子（如果大家不喜欢它，还请多多谅解——你可以想象一下它过去应该有多么糟糕）。因此，请允许我向这些人表示我的谢意，他们就是：Alejandro Aragón、Bill Baxter、Kevin Bealer、Travis Boucher、Mike Casinghino、Àlvaro Castro Castilla、Richard Chang、Don Clugston、Stephan Dilly、Karim Filali、Michel Fortin、David B. Held、Michiel Helvensteijn、Bernard Helyer、Jason House、Sam Hu、Thomas Hume、Graham St.Jack、Robert Jacques、Christian Kamm、Daniel Keep、Mark Kegel、Sean Kelly、Max Khesin、Simen Kjaeraas、Cody Koeninger、Denis Koroskin、Lars Kyllingstad、Igor Lesik、Eugene Letuchy、Pelle Mansson、Miura Masahiro、Tim Matthews、Scott Meyers、Bartosz Milewski、Fawzi Mohamed、Ellery Newcomer、Eric Niebler、Mike Parker、Derek Parnell、Jeremie Pelletier、Pablo Ripolles、Brad Roberts、Michael Rynn、Foy Savas、Christof Schardt、Steve Schveighoffer、Benjamin Shropshire、David Simcha、Tomasz Stachowiak、Robert Stewart、Knut Erik Teigen、Cristian Vlăsceanu、Leor Zolman。

Andrei Alexandrescu

2010 年 5 月 2 日，星期日

自在读书 www.zizidiary.com

目 录

第 1 章 进入 D 世界 …… 1

1.1 数字与表达式 …… 2
1.2 语句 …… 4
1.3 函数基础 …… 5
1.4 数组与关联数组 …… 6
1.4.1 构建词汇表 …… 6
1.4.2 数组分割、泛型函数、单元测试 …… 8
1.4.3 频率统计、λ 函数 …… 9
1.5 基本数据结构 …… 11
1.6 接口和类 …… 15
1.7 值与引用 …… 19
1.8 小结 …… 20

第 2 章 基本类型、表达式 …… 21

2.1 符号 …… 22
2.2 字面量 …… 23
2.2.1 布尔字面量 …… 23
2.2.2 整数字面量 …… 23
2.2.3 浮点数字面量 …… 24
2.2.4 字符字面量 …… 26
2.2.5 字符串字面量 …… 26
2.2.6 数组与关联数组字面量 …… 29
2.2.7 函数字面量 …… 30
2.3 操作符 …… 31
2.3.1 左值和右值 …… 31
2.3.2 隐式数值转换 …… 31
2.3.3 数值操作符的类型 …… 34
2.3.4 基本表达式 …… 34
2.3.5 后缀表达式 …… 37
2.3.6 一元表达式 …… 38
2.3.7 幂表达式 …… 41
2.3.8 乘法表达式 …… 41
2.3.9 加法表达式 …… 41
2.3.10 移位表达式 …… 41
2.3.11 in 表达式 …… 42
2.3.12 比较操作符 …… 43
2.3.13 按位操作符 …… 44
2.3.14 逻辑与 …… 45
2.3.15 逻辑或 …… 45
2.3.16 条件操作符 …… 45
2.3.17 赋值表达式 …… 46
2.3.18 逗号表达式 …… 46
2.4 小结和快速参考 …… 46

第 3 章 语句 …… 49

3.1 表达式语句 …… 49
3.2 复合语句 …… 50
3.3 if 语句 …… 50
3.4 static if 语句 …… 51
3.5 switch 语句 …… 53
3.6 final switch 语句 …… 54
3.7 循环语句 …… 55
3.7.1 while 语句 …… 55
3.7.2 do-while 语句 …… 55
3.7.3 for 语句 …… 55
3.7.4 foreach 语句 …… 56

自在读书 www.zizidiary.com

3.7.5 数组 foreach ····· 57
3.7.6 continue 语句和 break 语句 ····· 58
3.8 goto 语句 ····· 59
3.9 with 语句 ····· 60
3.10 return 语句 ····· 61
3.11 throw 语句和 try 语句 ····· 61
3.12 mixin 语句 ····· 62
3.13 scope 语句 ····· 63
3.14 synchronized 语句 ····· 66
3.15 asm 语句 ····· 67
3.16 小结和快速参考 ····· 67

第 4 章 数组、关联数组和字符串 ····· 70

4.1 动态数组 ····· 70
4.1.1 长度 ····· 71
4.1.2 边界检查 ····· 72
4.1.3 分割 ····· 73
4.1.4 复制 ····· 73
4.1.5 相等性比较 ····· 74
4.1.6 拼接 ····· 75
4.1.7 数组表达式 ····· 75
4.1.8 收缩 ····· 76
4.1.9 扩张 ····· 77
4.1.10 对 .length 赋值 ····· 79
4.2 固定大小的数组 ····· 80
4.2.1 长度 ····· 81
4.2.2 边界检查 ····· 81
4.2.3 分割 ····· 81
4.2.4 复制和隐式转换 ····· 82
4.2.5 相等性比较 ····· 82
4.2.6 拼接 ····· 83
4.2.7 数组运算 ····· 83
4.3 多维数组 ····· 83
4.4 关联数组 ····· 85
4.4.1 长度 ····· 85
4.4.2 读写槽 ····· 86
4.4.3 复制 ····· 86
4.4.4 相等性比较 ····· 86
4.4.5 移除元素 ····· 87
4.4.6 迭代 ····· 87
4.4.7 将自定义类型作为键 ····· 88
4.5 字符串 ····· 88
4.5.1 代码点 ····· 88
4.5.2 编码 ····· 89
4.5.3 字符类型 ····· 90
4.5.4 字符数组 + 优势 = 字符串 ····· 91
4.6 数组的远房表亲：指针 ····· 93
4.7 小结和快速参考 ····· 95

第 5 章 数据与函数、函数式 ····· 98

5.1 编写一个简单函数并进行单元测试 ····· 98
5.2 传递约定和存储类别 ····· 100
5.2.1 ref 参数与返回 ····· 101
5.2.2 in 参数 ····· 102
5.2.3 out 参数 ····· 102
5.2.4 static 数据 ····· 103
5.3 类型参数 ····· 103
5.4 签名约束 ····· 105
5.5 重载 ····· 106
5.5.1 偏序函数 ····· 108
5.5.2 跨模块重载 ····· 110
5.6 高阶函数、函数字面量 ····· 111
5.7 嵌套函数 ····· 113
5.8 闭包 ····· 114
5.9 超越数组、范围、伪成员 ····· 116
5.9.1 伪成员和 @property 属性 ····· 117
5.9.2 reduce——不只是证法 ····· 119
5.10 可变参数函数 ····· 120
5.10.1 同构可变参数函数 ····· 120
5.10.2 异构可变参数函数 ····· 121

自在读书 www.zizidiary.com

5.11 函数属性……124
5.11.1 纯函数……124
5.11.2 函数属性 nothrow……127
5.12 编译时计算……127

第 6 章 类、面向对象风格……132

6.1 类……132
6.2 对象名就是引用……133
6.3 对象的生命周期……136
6.3.1 构造函数……137
6.3.2 转发构造函数……138
6.3.3 构造步骤……139
6.3.4 析构与解除分配……140
6.3.5 拆毁步骤……141
6.3.6 静态的构造函数与析构函数……142
6.4 方法和继承……143
6.4.1 术语大杂烩……144
6.4.2 继承就是子类型化，静态类型和动态类型……145
6.4.3 改写只能是自愿的……146
6.4.4 调用被改写的方法……147
6.4.5 协变返回类型……148
6.5 静态成员的类级封装……149
6.6 使用 final 方法来限制扩展性……149
6.7 封装……151
6.7.1 private……152
6.7.2 package……152
6.7.3 protected……152
6.7.4 public……152
6.7.5 export……153
6.7.6 封装到什么程度……153
6.8 一统天下的根类……155
6.8.1 string toString()……156
6.8.2 size_t toHash()……156
6.8.3 bool opEquals(Object rhs)……156
6.8.4 int opCmp(Object rhs)……159
6.8.5 static Object factory (string className)……160
6.9 接口……161
6.9.1 非虚接口（NVI）惯用法……162
6.9.2 protected 原语……164
6.9.3 选择性实现……165
6.10 抽象类……166
6.11 嵌套类……168
6.11.1 函数嵌套类……170
6.11.2 静态嵌套类……171
6.11.3 匿名类……172
6.12 多重继承……172
6.13 多重子类型化……175
6.14 参数化的类和接口……177
6.15 小结……180

第 7 章 其他自定义类型……181

7.1 结构……182
7.1.1 复制语义……183
7.1.2 将结构对象传递给函数……184
7.1.3 结构对象的生命周期……184
7.1.4 静态构造函数和析构函数……192
7.1.5 方法……193
7.1.6 静态成员……197
7.1.7 访问指示符……197
7.1.8 嵌套结构和类……198
7.1.9 在函数内部嵌套结构……199
7.1.10 结构子类型化与@disable 属性……199
7.1.11 字段布局与对齐……202
7.2 联合……204
7.3 枚举值……206
7.3.1 枚举类型……208
7.3.2 枚举特性……208
7.4 别名……209

自在读书 www.zizidiary.com

7.5 使用模板实现参数化作用域……211
7.6 使用 mixin 模板注入代码……214
7.7 小结与参考……216

第 8 章 类型修饰符……217

8.1 immutable 修饰符……218
8.2 immutable 的书写……220
8.3 immutable 参数和方法……221
8.4 immutable 构造函数……222
8.5 immutable 相关的转换……223
8.6 const 修饰符……225
8.7 const 与 immutable 进行交互……226
8.8 将修饰符从参数传送至结果……226
8.9 小结……227

第 9 章 错误处理……229

9.1 抛出与捕捉……229
9.2 类型……230
9.3 finally 子句……232
9.4 nothrow 函数和 Throwable 的特殊本质……233
9.5 附属异常……234
9.6 栈展开与异常安全代码……235
9.7 未捕捉的异常……237

第 10 章 契约编程……238

10.1 契约……239
10.2 断言……240
10.3 前置条件……241
10.4 后置条件……243
10.5 不变量……244
10.6 跳过契约检查与发行版本……246
10.6.1 enforce 不（完全）是 assert……247
10.6.2 assert(false)……248
10.7 契约：请不要用于洗涤输入……248
10.8 契约与继承……250
10.8.1 继承与 in 契约……250
10.8.2 继承与 out 契约……252
10.8.3 继承与 invariant 契约……253
10.9 接口里的契约……253

第 11 章 扩展性……255

11.1 包和模块……255
11.1.1 import 声明……256
11.1.2 模块搜索根目录……257
11.1.3 名字查找……258
11.1.4 public import 声明……260
11.1.5 static import 声明……261
11.1.6 选择性导入……262
11.1.7 在导入里重新命名……262
11.1.8 module 声明……264
11.1.9 模块摘要……264
11.2 安全性……267
11.2.1 定义行为和未定义行为……267
11.2.2 @safe、@trusted 和 @system 属性……268
11.3 模块的构造函数与析构函数……269
11.3.1 模块内的执行顺序……270
11.3.2 跨模块的执行顺序……270
11.4 文档注释……271
11.5 与 C 和 C++进行交互……271
11.6 deprecated 声明……272
11.7 version 声明……272
11.8 debug 声明……273
11.9 D 语言标准库……273

第 12 章 操作符重载……275

12.1 重载操作符……276
12.2 重载一元操作符……276
12.2.1 使用 mixin 加强操作符定义……277
12.2.2 后增值与后减值……278

自在读书 www.zizidiary.com

12.2.3 重载 cast 操作符……278
12.2.4 三元操作符重载测试和 if 测试……279
12.3 重载二元操作符……280
12.3.1 操作符重载的平方……281
12.3.2 交换律……281
12.4 重载比较操作符……282
12.5 重载赋值操作符……283
12.6 重载索引操作符……284
12.7 重载分割操作符……285
12.8 $操作符……286
12.9 重载 foreach……286
12.9.1 带有迭代原语的 foreach……286
12.9.2 带有内部迭代的 foreach……287
12.10 在类里定义重载操作符……289
12.11 完全不一样的重载：opDispatch……290
12.12 小结与快速参考……292

第 13 章 并发性……294

13.1 并发门……295
13.2 数据共享简史……296
13.3 快瞧瞧，没有（默认）共享……299
13.4 启动线程……300
13.5 线程间的信息交换……302
13.6 receive 的模式匹配……303
13.6.1 首次匹配……305
13.6.2 匹配所有消息……305
13.7 文件复制——稍有改变……305
13.8 线程终止……307
13.9 带外通信……308
13.10 邮箱堵塞……309
13.11 shared 类型修饰符……310
13.12 shared 数据操作及其效果……312
13.13 使用 synchronized 类实现基于锁的同步……313
13.14 synchronized 类里的字段类型化……316
13.14.1 临时保护==无脱离……317
13.14.2 局部保护==尾部共享……317
13.14.3 强制同质的互斥量……319
13.14.4 不可思议：shared 可被强制转换掉……320
13.15 死锁与 synchronized 语句……320
13.16 使用 shared 类进行无锁编码……321
13.16.1 shared 类……322
13.16.2 几种无锁结构……323
13.17 小结……325

参考文献……327

自在读书 www.zizidiary.com

第1章

进入D世界

大家都知道该从哪里开始，那我们就不废话了，请看下面的代码：

```
import std.stdio;
void main() {
    writeln("Hello, world!");
}
```

如果你对其他语言也有所了解，那么是否会觉得这个特别简单呢？甚至可能还会有一点失望：D 语言走的并不是脚本语言的路线，也不像它们那样支持语句至上的原则。语句至上原则会引入很多全局变量。随着程序规模的变大，这些全局变量会迅速变成一种累赘；而 D 语言提供的是一种在 main 之外执行代码的能力，而且提供的方式更结构化。如果大家对精度的要求很严格，那么请放心：void main 与 int main 其实是等效的。如果执行过程顺利结束，它们都会向操作系统返回“成功”（代码为 0）标识。

我们先不着急往前推进。传统的“Hello, world!”程序不是用来讨论一门语言所拥有的功能的，而是让大家可以从一开始就使用这种语言来编写并运行程序。如果大家没有合适的能提供透明构建功能的 IDE，那么选用命令行的方法也不会太难。将上述代码录入到一个文件，如 hello.d，运行控制台 shell，然后输入下面的命令：

```
$ dmd hello.d
$ ./hello
Hello, world!
$ _
```

其中，$表示的是命令提示符（在 Windows 系统中，它是“C:\Path\To\Dir>”；在 OSX、Linux 和 Cygwin 这种类 Unix 的系统中，它是“/path/to/dir %”）。只需大家应用一点点操作系统技巧，就能让程序自动编译和运行起来。在 Windows 系统里，大家或许会想到将 shell 命令 run 与程序 rdmd.exe 联系起来。对于类 Unix 系统，由于它们支持使用“shebang 符号”（井号加叹号）方式来启动脚本，而且 D 语言也支持这种语法，因此将下面一行添加到程序 hello.d 的最开始位置便能直接运行它。

```
#!/usr/bin/rdmd
```

自在读书
www.zizidiary.com

在完成如上修改后，只需要在命令提示符处简单地输入下面的内容（chmod 命令只需要执行一次）：

```
$ chmod u+x hello.d
$ ./hello.d
Hello, world!
$ _
```

在所有操作系统里，程序 rdmd 都表现得非常智能。它能够缓存已生成的可执行文件，因此重新编译只会针对被更改过的程序，并不需要每次运行它时也编译一次。最终结果就是整个编译过程非常迅速，从而形成一个快速的“编辑-运行”周期——这一点对于简短的脚本和大型程序都同样有效。

应用程序自身是从下面指令开始的：

```
import std.stdio;
```

该指令会指示编译器去搜寻一个名叫 std.stdio 的模块，并解析出它所提供的符号。import 相当于C和C++语言里的#include 预处理指令，但在语义上却更接近于Python 语言的 import——不会产生文本包含，获取到的仅仅是一张符号表。重复导入同一文件并不会产生新的导入。

遵照 C 语言所创立的受人敬重的传统，D 语言程序也是由一组遍布于多个文件的声明所构成的。这些声明还可能会引入其他的内容，如类型、函数和数据等。我们的第一个程序就定义了一个不带任何参数的主函数——main，该函数返回的是“空内容”，即 void。当 main 函数被调用时，它又会去调用 writeln 函数（当然，此函数已事先在 std.stdio 模块里进行了定义），同时还会向该函数传递一个字符串常量。后缀 ln 表示 writeln 函数会在输出文本里添加一个换行符。

下面的内容是对 D 语言世界的一个快速浏览。为说明 D 语言的基本概念，会引入一些小段的示例程序。其重点是想让大家尽快建立起对 D 语言的感性认识，而不是要进行刻板的定义。后面的章节则会对语言中的各个部分进行更详细的说明。

1.1　数字与表达式

外国人有多高？大家是否有过这样的好奇呢？现在我们来编写一个简单的程序，它实现的是用“英尺+英寸”和“厘米”两种单位显示一系列常见身高值。

```
/*
  将一系列以英尺和英寸两种单位表示的身高值转换成以厘米为单位的值
*/
import std.stdio;

void main() {
    // 不太可能变化的值
    immutable inchesPerFoot = 12;
    immutable cmPerInch = 2.54;

    // 循环输出
    foreach (feet; 5 .. 7) {
        foreach (inches; 0 .. inchesPerFoot) {
            writeln(feet, "'", inches, "''\t",
                (feet * inchesPerFoot + inches) * cmPerInch);
        }
    }
}
```

自在读书 www.zizidiary.com

执行此程序，会输出一个列表，它含有两个整齐的列：

```
5'0''    152.4
5'1''    154.94
5'2''    157.48
...
6'10''   208.28
6'11''   210.82
```

`foreach (feet; 5 .. 7) { ... }`是一个迭代语句，它定义了一个整型变量 `feet`，并依次将其设置为 5 和 6。请注意，不包括 7，这是一个右开区间。

与 Java、C++和 C#一样，D 语言也支持“`/*多行注释*/`”和“`//单行注释`”两种注释风格（另外还支持文档注释，这个会在后面说明）。这里还有一个很有意思的细节，即我们的小程序引入数据的方式。首先，它定义了两个常量：

```
immutable inchesPerFoot = 12;
immutable cmPerInch = 2.54;
```

这些带有关键字 `immutable` 的常量绝对不会被更改。跟变量一样，常量也可以不必提供明确的类型说明，编译器会根据该符号的初始值推断出实际的类型。因此，文字 `12` 其实是告知编译器：`inchesPerFoot` 是一个整型变量（在 D 语言里使用 `int` 来表示）；同样地，文字 `2.54` 会让 `cmPerInch` 变成一个浮点常量（类型为 `double`）。继续往前，我们还会注意到，`feet` 和 `inches` 的定义也充满了同样的魔法，因为它们表面上是变量，却没有明确的类型修饰。与下面的做法相比，那样做并没有让程序变得不安全；恰好相反，它让程序变得更加简洁：

```
immutable int inchesPerFoot = 12;
immutable double cmPerInch = 2.54;
...
foreach (int feet; 5 .. 7) {
    ...
}
```

只有当根据上下文能够进行无歧义的类型推导时，编译器才允许忽略类型声明。刚才我们提到了类型，那就先暂停一下，一起来看看可以使用的数字类型有哪些。

按大小递增的顺序，有符号整数类型包括 `byte`、`short`、`int` 和 `long`，它们对应的大小分别为 8、16、32 和 64 位。其中每一种类型还对应有一个同样大小的无符号类型，即 `ubyte`、`ushort`、`uint` 和 `ulong`。（在 C 语言里并没有“无符号”修改符。）浮点类型包括 `float`（32 位 IEEE 754 单精度数）、`double`（64 位 IEEE 754）和 `real`（它与机器的浮点寄存器允许的范围一样大，但不会小于 64 位；例如，在 Intel 机器上，`real` 即为所谓的 IEEE 754 扩展双精度 79 位格式）。

现在回到稳健的整数领域，像 `42` 那样的字面量可以赋给任何数字类型，只不过编译器会检查目标类型所允许的范围大小，看它能否容纳下该值。下面的声明与省略掉 `byte` 的声明效果一样：

```
immutable byte inchesPerFoot = 12;
```

这是因为对于 `12` 来说，8 位与 32 位类型都可以用来表示。默认情况下，如果目标类型需要由数字进行推导（如这里的示例），那么整型常量的类型就是 `int`，而浮点常量的类型则为 `double`。

运用这些类型，再借助算术操作符和函数，就能够在 D 语言里构建出大量的表达式。操作符及其优先级都可以从其他与 D 语言同族的语言里找到，如`+`、`*`、`/`和`%`用于基本的算术运算，`==`、`!=`、`<`、`>`、`<=`、`>=`用于比较运算，`fun(argument1, argument2)`用于函数调用等。

自在读书 www.zizidiary.com

再回到那个英寸转换为厘米的程序，这里有两个关于调用 writeln 函数的细节需要注意。一个细节是 writeln 接受了 5 个参数（它与“Hello, world!”程序里的那个函数完全不一样）。这跟其他语言的 I/O 操作非常相似，如 Pascal 语言的 writeln、C 语言的 printf 或者 C++ 语言的 tout，D 语言的 writeln 函数能够接受个数可变的参数，即它是一个“可变参数函数”（variadic function）。在 D 语言里，用户也可以定义自己的可变参数函数（这一点跟 Pascal 语言有所不同），而且这些函数还总是类型安全的（这一点跟 C 语言又有所不同）。另一个细节是对 writeln 的调用将格式信息与要被格式化的数据混合在一起，而且很难看。人们总是希望能将数据与呈现分离开来，因此这里也可以使用格式化的输出函数 writefln：

```
writefln("%s'%s''\t%s", feet, inches,
   (feet * inchesPerFoot + inches) * cmPerInch);
```

调整后的调用输出了完全一样的结果，前后的差别在于 writefln 的第一个参数完整地对格式进行了描述。与 C 语言的 printf 一样，%会引入一个格式说明符，例如：%d 代表的是整数，%f 代表的是浮点数，而%s 代表的是字符串。

如果大家使用过 printf，那么可能会有一种亲切感，不过还有一个特别小的细节需要注意一下：我们要输出的是整数和双精度数，可它们怎么都使用了格式说明符%s 来进行格式描述，难道它不只是用于字符串描述吗？答案很简单。D 语言的可变参数功能让 writefln 可以访问到传入参数的实际类型，其中包含两层意思：（1）%s 的含义会被扩展为“任何参数的默认字符串表示”；（2）如果参数的实际类型与该说明符不匹配，则会给出一个明确的错误，而不会试图使用怪异的说明符去对 printf 调用进行错误的格式化（关于调用 printf 时使用不可信格式字符串所导致的潜在安全漏洞，这里就不多说了）。

1.2　语句

跟同族的语言一样，在 D 语言里，后面紧跟有一个分号的任何表达式就是一条语句，如“Hello world!”程序里的 writeln 调用，它后面就带有一个;。语句的作用是要简化表达式的计算。

D 语言也是“花括号块作用域”一族的成员，即可以通过“{”和“}”将多条语句括起来形成一组语句——这是很有必要的，例如需要在一个 foreach 循环里完成多件事情。如果恰好只有一条语句，则可以完全省略花括号。事实上，那个执行身高转换的双重循环可以改写成下面的样子：

```
foreach (feet; 5 .. 7)
  foreach (inches; 0 .. inchesPerFoot)
    writefln("%s'%s''\t%s", feet, inches,
      (feet * inchesPerFoot + inches) * cmPerInch);
```

对于单条语句，省略花括号的好处在于可以缩短代码，而坏处就是会让编辑变得比较费事（在代码维护阶段，随着语句的调整，通常都需要添加或删除花括号）。人们习惯利用缩进和花括号来实现整齐分隔。事实上，只要你保持一致，这种事情也并不是那么重要。举个例子，因为排版的关系，本书所使用的样式（即使对于单条语句也使用了括号来将其围住：正括号位于开始行，反括号则独立占一行）就与作者平日里代码的样式有很大不同。如果不会因这样做而

自在读书 www.zizidiary.com

变成一个“狼人”的话，那就继续吧。

Python 语言采用了一种与众不同的样式——形式遵从结构（form follows structure），它通过缩进的方式恰如其分地来表示块结构，空白符也要起作用。这对于其他语言的程序员来说有些奇特，但 Python 程序员一直坚持这一点。D 语言在通常情况下会将空白符完全忽略掉，不过也有进行特别地设计，以便更好地解析它们（如解析时不需要去理解这些符号的意义）。因此，一个好的项目应该实现一个简单的处理程序，以便在 D 语言里也支持使用 Python 语言的缩进样式，进而不用去忍受编译、运行和调试程序过程中出现的种种不便。

`if` 语句的一般形式如下：

```
if (<表达式>) <语句1> else <语句2>
```

有个名叫“构造定理”（见参考文献[10]）的理论结论，它证明我们只使用复合语句就能够实现任何算法：使用 `if` 进行测试，而循环可以使用 `for` 和 `foreach`。当然任何实际语言提供的内容远不止这些，D 语言也不例外，一起继续往下看吧。

1.3　函数基础

我们先跳过必不可少的 `main` 函数定义，一起来看看在 D 语言里如何定义其他的函数。函数定义遵从的模式也可以从其他与 Algol 类似的语言里找到：首先是返回类型，接着是函数名称，最后是形式参数（formal parameter）[①]——它是一个使用圆括号括起来的以逗号隔开的列表。例如，想要定义一个名叫 `pow` 的函数，它接受一个 `double` 和一个 `int` 类型，并返回一个 `double`，则可以写成这样：

```
double pow(double base, int exponent) {
    ...
}
```

每一个函数参数（在上面的示例中为 `base` 和 `exponent`）都附带有一个类型，除此之外，还可以额外添加一个*存储类别*（storage class），用以声明函数被调用时参数传递到函数内部的方式。默认情形下，实参是通过值的方式进行传递的。如果在某个形参的类型里使用了存储类别 `ref`，那么该形参就会直接绑定到输入实参上；这样，对形参的更改便会立即直接反映到与之对应的外部实参上。举例：

```
import std.stdio;

void fun(ref uint x, double y) {
    x = 42;
    y = 3.14;
}
void main() {
    uint a = 1;
    double b = 2;
    fun(a, b);
    writeln(a, " ", b);
}
```

上面的程序会输出 `42 2`。这是因为 `x` 类型为 `ref uint`，意味着对 `x` 的赋值其实是赋给了 `a`。

① 本书会固定使用“形参”（parameter）来暗指在函数内部接受到的和使用的值，而在谈论函数调用期间从外部传递到函数的值时，会使用“实参”（argument）。

自在读书 www.zizidiary.com

而同时，对 y 的赋值并没有对 b 产生影响，这是因为函数 fun 将 y 处理成了一个私有副本。

在本段介绍的最后要讨论一下装饰类型是 in 和 out。简单地讲，函数里的 in 用于保证参数只会进行查看，而不会被更改。在函数参数里使用 out 的效果近似于 ref，不同之处在于参数会在进入到函数时被初始化成默认值。每一个类型 T 都会定义一个初始值，其形式为 T.init。用户自定义类型可以定义自己的 init。

关于函数还有很多值得说的。你可以将函数传递给其他函数，可以将它们彼此嵌套在一起，可以让函数保存自己的本地环境（完备的语法闭包），可以创建并随意地操作匿名函数（λ 函数）以及其他一些有用的小功能。这些会在后面依次进行讨论。

1.4　数组与关联数组

数组与关联数组（后者也就是大家常说的散列表或散列）可能是计算史上最常用到的复合数据结构，其次要算是 Lisp 的列表。大量的程序需要的无非就是某种类型的数组和关联数组，因此现在一起来看看 D 语言是如何实现它们的。

1.4.1　构建词汇表

例如，按照下面的要求编写一个简单的程序：

读入一段由使用空白符隔开的单词所组成文本，并为不同的单词关联一个唯一数字。最后，以“标识 单词”的形式输出各行。

这段小脚本在进行文本处理时相当有用：一旦建立好词汇表，就只需要去操作那些数字（代价更小），从而避开对所有单词的操作。有一种构建此类词汇表的方法就是：在一个将单词映射成整数的词典里，按照单词出现的顺序进行累计。当添加新的映射时，我们只需要确保该整数是唯一的就行（比较可靠的方法就是只使用词典的当前长度，最后标识就是 0，1，2，…）。现在一起来看看使用 D 语言会怎么做。

```
import std.stdio, std.string, std.algorithm;

void main() {
    uint[string] dictionary;
    foreach (line; stdin.byLine()) {
        // 将句子分成单词组，并将每一个单词添加到词汇表
        foreach (word; splitter(strip(line))) {
            if (word in dictionary) continue; // 什么也不做
            auto newID = dictionary.length;
            dictionary[word.idup] = newID;
            writeln(newID, '\t', word);
        }
    }
}
```

在 D 语言里，假如关联数组（散列表）实现的功能是将类型 K 映射到类型 V，则其类型需被标识成 V[K]。因此，类型为 uint[string]的变量 dictionary 会将字符串映射到无符号整数——正是我们所需要的，可用于存储“单词-标识”映射关系。如果关键字 word 能在关联数

自在读书 www.zizidiary.com

组词典中查找到，则表达式 word in dictionary 的结果将是一个非零值。最后，通过 dictionary[word] = newID 来完成新单词的插入。

尽管在上面的脚本中未做明确说明，但类型 string 实际上就是一个字符数组。一般情况下，类型为 T 的动态大小数组可以表示成 T[]，并且存在多种分配方式，如：

```
int[] a = new int[20]; // 20 个整数，初始值为 0
int[] b = [ 1, 2, 3 ]; // 包含元素 1、2 和 3 的数组
```

与 C 语言数组不同，D 语言的数组知道自己的长度，例如 arr.length 访问的就是数组 arr 的长度。对 arr.length 赋值会导致数组重新分配。数组访问是带边界检查。如果有代码不怕承担缓冲区溢出的风险，也可以将数组的指针提取出来（方法是使用 arr.ptr），然后使用无边界检查的指针运算。另外，如果的确还想要“硅片”提供的所有内容，那么就使用编译器选项来禁用边界检查。这样就为通往安全性铺平了道路：默认情况下，代码是安全的；然而也可以通过付出更多的努力来获得微量的提速。

下面展示的是如何使用已经熟悉的 foreach 语句的新形式实现数组的迭代操作：

```
int[] arr = new int[20];
foreach (elem; arr) {
   /* ... 使用 elem ...*/
}
```

上面的循环将 elem 依次绑定到了数组 arr 的每一个元素。给 elem 进行赋值并不会影响到 arr 里的元素。想要更改数组，只需要使用关键字 ref 即可：

```
// 将 arr 的所有元素置 0
foreach (ref elem; arr) {
   elem = 0;
}
```

现在我们已了解 foreach 是如何处理数组的，接下来再进一步，看一个更有用的操作。如果在进行迭代时，还需要数组元素的索引值，依然可以使用 foreach：

```
int[] months = new int[12];
foreach (i, ref e; months) {
    e = i + 1;
}
```

上面的代码会创建一个包含 1, 2, …, 12 的数组。该循环与下面稍显冗长的代码具有同样的效果。下面的代码是使用 foreach 来迭代一段数字：

```
foreach (i; 0 .. months.length) {
   months[i] = i + 1;
}
```

D 语言也提供有静态大小的数组，其表示形式类似于 int[5]。除了少量有特殊需求的程序需要它以外，应该首选动态大小的数组，这是因为多数情形下事先并不知道数组的确切大小。

数组具有浅复制语义，即将一个数组变量复制到另一个数组时，并不会复制整个数组，仅仅是针对同一个底层存储产生一个新的视图。如果想得到副本，只需要使用数组的 dup 特性即可。

```
int[] a = new int[100];
int[] b = a;
// ++x 会递增 x 的值
++b[10];      // b[10]现在为 1，而 a[10]也一样
b = a.dup;    // 将 a 整个复制到 b
++b[10];      // b[10]现在为 2，而 a[10]仍为 1
```

自在读书 www.zizidiary.com

1.4.2　数组分割、泛型函数、单元测试

数组分割（array slicing）是一项强大的功能，它允许引用数组的某部分，而无需进行数组数据的复制。举例来说，现编写一个函数 binarySearch 来实现二分搜索算法：给定一个有序数组和一个值，binarySearch 要快速地返回一个布尔值，以表明该值是否存在于此数组当中。D 语言的标准库提供了一个函数以一种更通用的方式实现这个功能，并且会返回更多的信息，而不仅仅是一个布尔值。不过实现它需要了解更多的 D 语言功能。把我们的目标提高那么一点，让编写出的 binarySearch 不仅可用于整数数组，还能用于其他任何类型的数组，只需要该类型支持使用“<”进行比较操作。实现它其实也不会花费太多时间，泛型版的 binarySearch 函数如下：

```
import std.array;

bool binarySearch(T)(T[] input, T value) {
   while (!input.empty) {
      auto i = input.length / 2;
      auto mid = input[i];
      if (mid > value) input = input[0 .. i];
      else if (mid < value) input = input[i + 1 .. $];
      else return true;
   }
   return false;
}

unittest {
   assert(binarySearch([ 1, 3, 6, 7, 9, 15 ], 6));
   assert(!binarySearch([ 1, 3, 6, 7, 9, 15 ], 5));
}
```

在函数 binarySearch 的签名[①]处有一个记号(T)，它会引入一个类型参数 T。类型参数可以在函数的常规参数列表里使用。当 binarySearch 被调用时，它会根据接收到的实参对 T 进行推导。如果想显式指定 T（如想要达到双重检查的目的），则可以编写出类似下面的内容：

```
assert(binarySearch!(int)([ 1, 3, 6, 7, 9, 15], 6));
```

它显露出来的信息是一个泛型函数需要使用两对括号参数来进行调用。首先使用!(...)括起来的编译时参数，接着使用(...)括起来的运行时参数。将分属两个范畴的内容混合在一起的做法是经过深思熟虑的，不过实验表明这种统一形式带来的麻烦比它消除的还要多。

如果你熟悉 Java、C#或 C++里类似的功能，自然会注意到 D 语言与这些使用尖括号<和>来指定编译时参数的语言有着明显的差异。这是一个人为的决定，其目的在于想尽量避免由 C++语言的经验所显露的高额代价，如解析难度的增加、大量的特殊规则和随意决断以及用户难辨的晦涩语法[②]。此难题的根源在于<和>事实上同时也是比较操作符[③]。如果把它们当作分隔符，当允许表达式插入这些分隔符内部时，很容易产生歧义。由此导致这些分隔符很难进行配对。

① 函数签名（function signature）是由函数的原型构成，它所提供的是关于函数的一般性信息，如函数名、参数、函数所处的作用域以及其他一些信息，具体请参考 http://www.cs.unm.edu/~storm/C++/Programming Terms/Function Signatures.html。——译者注

② 如果你认识一个拥有超人能力的 C++编码员，请问问他像 object.templatefun<arg>()这样的语法是干什么用的，然后你就能看到氪气石（译注：超人故事里的神奇矿物）发挥作用了。

③ 再“撒点儿盐”，<<和>>也都是运算符。

自在读书 www.zizidiary.com

Java 和 C#之所以没这种问题，原因在于它们不允许“<”和“>”内部出现表达式。不过这种做法让它们为获得某些可能存在的好处，而限制了自己以后的扩展能力。将传统的一元操作符“!”扩展成二元操作符来使用，同时使用传统的圆括号（这个保证你的配对总是正确的），D 语言通过这种方法便能支持将表达式当做编译时参数来使用，并且由此简化人和编译器的工作。

在 binarySearch 的实现里，另外一个有意思的细节是使用 auto 来实现类型的推断：i 和 mid 的类型都是根据它们的初始化表达式推断出来的。

为保持良好的编程习惯，binarySearch 还附带有一个单元测试。单元测试就是一块语句，其前缀是关键字 unittest（一个文件可以根据需要包含多个单元测试，你明白这意味着什么——几乎多到满额）。将参数标志-unittest 传递给编译器，便可以实现在进入 main 函数之前运行单元测试。虽然 unittest 看起来像是一个小功能，不过它让插入小测试的操作变得很容易（如果达不到这一点，则让人有些尴尬），从而帮你实现良好的编程风格。当然，如果你是处于高端的思想家，喜欢先看到单元测试，再看到具体实现，那么也可以很轻松地实现将单元测试挪到 binarySearch 的前面。在 D 语言里，处于模块级的符号，其语义绝不会依赖于与其他符号的相对顺序。

分割表达式 input[a .. b] 会返回 input 的一个分片，其范围从索引值 a 直至索引值 b（但不包括它）。如果有 a == b，那么将会生成一个空的分片；而如果有 a > b，则会抛出一个异常。一次分割并不会引起动态内存分配，它仅仅是该数组里某一部分的一个别名。在索引表达式或分割表达式里，$代表的是正被访问的那个数组的长度。例如，input[0 .. $] 与 input 代表的其实是一个内容。

虽然 binarySearch 看起来移动了大量的数组，但根本就不会有数组重新分配。所有 input 的分片都与原始的那个 input 共享同一个内存空间。与传统使用索引值的实现相比，此实现的运行效率绝不可能更低，但它却更容易理解，因为它操作的状态更少。说到状态，现在我们一起来编写一个 binarySearch 的递归实现，它完全没有对 input 重新赋值：

```
import std.array

bool binarySearch(T)(T[] input, T value) {
    if (input.empty) return false;
    auto i = input.length / 2;
    auto mid = input[i];
    if (mid > value) return binarySearch(input[0 .. i], value);
    if (mid < value) return binarySearch(input[i + 1..$], value);
    return true;
}
```

与相应的迭代实现相比，此递归实现毫无疑问更简洁。它也一样地很高效，这是因为有一项流行的编译器技术尾部调用消除（tail call elimination），可以很容易优化递归调用。关于尾部调用消除技术，扼要地讲就是，如果某个函数的 return 语句仅仅是使用了不同的参数来调用其自身，那么编译器会修改这些参数，并产生一个跳转到函数开始处的操作。

1.4.3　频率统计、λ 函数

现在来编写另一个有用的程序：统计文本里的不同单词数。想知道话剧《哈姆雷特》里哪一个单词的使用次数最多吗？马上就能实现。

下面的程序使用了一个关联数组将 string 类型映射到 uint 类型，而整个结构与前面的

自在读书 www.zizidiary.com

构建词汇表示例很接近。在添加一个简单的输出循环之后，一个有用的频率统计程序便完成：

```
import std.stdio, std.string;

void main() {
   // 计算统计
   uint[string] freqs;
   foreach (line; stdin.byLine()) {
      foreach (word; split(strip(line))) {
         ++freqs[word.idup];
      }
   }
   // 输出统计
   foreach (key, value; freqs) {
      writefln("%6u\t%s", value, key);
   }
}
```

一切就绪，现在上网访问链接 http://erdani.com/tdpl/hamlet.txt，下载文件 hamlet.txt，然后运行此程序便可得到如下输出：

```
     1    outface
     1    come?
     1    blanket,
     1    operant
     1    reckon
     2    liest
     1    Unhand
     1    dear,
     1    parley.
     1    share.
...
```

很遗憾，这里的输出并不是有序的，而且无论哪一个单词排在最前面都很难保证它恰好是频率最高的。这一点都不奇怪，为了让实现的基本操作执行起来尽可能快，关联数组允许在内部以任意顺序存储数据。

为将输出排序，使得最高频率的单词排在第一位，只需要将此程序的输出利用通道传送到 sort-nr（按数字反向排序）即可，不过这是一种“欺骗”方式。为将排序操作集成到程序里，我们需要将最后一个循环替换成下面的代码：

```
// 输出统计
string[] words = freqs.keys;
sort!((a, b) { return freqs[a] > freqs[b]; })(words);
foreach (word; words) {
   writefln("%6u\t%s", freqs[word], word);
}
```

特性.keys 产生的是关联数组 freps 的所有键值，在这里它就是一个字符串数组。此数组是重新分配的，这样做的原因在于我们需要调整字符串的顺序。现在来看看下面这段代码：

```
sort!((a, b) { return freqs[a] > freqs[b]; })(words);
```

它正好符合我们已提及的模式：

```
sort!(<编译时参数>)(<运行时参数>);
```

去掉一层圆括号!(...)之后，会得到与下面内容一样的一组符号——看起来像是一个不完整的函数，忘了提供参数类型、返回类型以及函数的名称：

自在读书 www.zizidiary.com

```
(a, b) { return freqs[a] > freqs[b]; }
```

它就是“λ 函数”——一种简短的匿名函数，通常用来传递给其他函数。λ 函数在很多地方都能派上用场，D 语言尽了最大的努力来剔除那些定义 λ 函数时不必要的语法包袱。参数类型跟返回类型一样都可以进行推断，这一点极具意义，因为 λ 函数的主体就明明白白地放置在那里，作者、读者以及编译器都可以看到，因而不会出现误解，也不会对模块化原则产生破坏。

关于在本示例里定义的 λ 函数，其中还有一个小细节需要说明一下。该 λ 函数访问了属于 main 函数局部（即是说，它不是全局的或静态的）的变量 freps。这项特性让 D 语言更像是 Lisp 语言，而非 C 语言，并且使 λ 函数的功能更为强大。尽管传统意义上这种功能伴随有运行时成本（需要间接的函数调用），D 语言却保证了不会出现间接调用（从而完全有了实现内联功能的机会）。

调整后的程序输出结果如下：

```
929 the
680 and
625 of
608 to
523 I
453 a
444 my
382 in
361 you
358 Ham.
...
```

正如所期望的，最常使用的单词也是频率最高的，不过“Ham.”除外。这并不表明剧中的主人公有强烈的烹饪爱好，它仅仅是所有哈姆雷特台词行的前缀。很显然，通篇他有 358 次需要进行说明，比其他任何人都多。如果向下浏览这个列表，就会发现下一个说话最多的人是国王，却只占了 116 行，还不到哈姆雷特的三分之一。而 Ophelia 则有些沉默寡言，仅占 58 行。

1.5　基本数据结构

既然我们已经得到了《哈姆雷特》，现在进一步对文本进行分析。例如，我们想收集一些关于所有剧中人物的信息，如他们总共说了多少个单词，以及他们的词汇量有多丰富。为实现这一点，需要将几个数据项目关联到每一个人物上去[①]。想要将这些信息组合到一起，可以定义一个类似下面那样的数据结构：

```
struct PersonaData {
   uint totalWordsSpoken;
   uint[string] wordCount;
}
```

在 D 语言里，可以使用结构（struct），也可以使用类（class）。它们有许多共同之处，但也存在一些差异：结构是值类型，而类表示的是动态多态性，并且它只能通过引用进行访问。这种方式让人有点困惑，与分割相关的错漏以及像“// 别这样！不能进行继承！”这样的注释都不存在。

① 很抱歉使用了“persona”一词。使用通俗的“character”一词的问题在于它可能造成与字符类型 char 之间的混淆。（在英文中 persona 和 character 都可以指人物，但后者容易跟字符类型 char 混淆，在中文里没有问题。——译者注）

自在读书 www.zizidiary.com

当设计一个类型时，首先需要弄清楚它是一个单值还是一个多态引用。C++语言支持定义“性别模糊”的类型，不过它们的用途很少、易出错且让人生厌，因此在设计时要尽量避免使用它们。

在这里，我们只想要收集一些数据，并没有过多的想法，因此使用结构会是一个不错的选择。现在来定义一个可以将人物名字映射到 PersonaData 值的关联数组：

```
PersonaData[string] info;
```

所有要做的事就是根据 hamlet.txt 来填充 info。这需要费些功夫，因为一个人物的段落可能占用了好几行，因此我们需要进行一些简单的处理来将多个物理行合并成段落。为弄明白怎么做，我们来看一下 hamlet.txt 里的一小段内容，原样排列如下（为看起更清晰一些，对顶头的空格符进行了标识）：

```
␣␣Pol. Marry, I will teach you! Think yourself a baby
␣␣␣␣That you have ta'en these tenders for true pay,
␣␣␣␣Which are not sterling. Tender yourself more dearly,
␣␣␣␣Or (not to crack the wind of the poor phrase,
␣␣␣␣Running it thus) you'll tender me a fool.
␣␣Oph. My lord, he hath importun'd me with love
␣␣␣␣In honourable fashion.
␣␣Pol. Ay, fashion you may call it. Go to, go to!
```

Polonius 关于“走吧[①]”的热情到底是不是致使他死亡的原因呢？即使到今天，其结果也仍然只是个猜测。我们不去深究他的死因，只关心一下每一个人物的台词行是什么样子：开始恰好是两个空格，紧跟着是人物名，然后是一个句点和一个空格，最后是行的实际内容。如果一个逻辑行扩展到了多个物理行，那么连续部分恰好是以四个空格作为引导。通过正则式引擎（在 std.regex 模块里可以找到）便可以完成这种简单的模式匹配，但我们要学习使用数组，因此这里通过人工实现匹配。我们只借助了布尔函数 a.startsWith(b)，它定义在 std.algorithm 模块里，能够识别出 a 是不是由 b 引导。

main 函数会读取所有输入行，将它们连接成逻辑段落（不符合模式的所有内容都忽略掉），将整个段落传递给累计函数，然后输出期望的信息。

```
import std.algorithm, std.conv, std.ascii, std.regex,
    std.range, std.stdio, std.string;

struct PersonaData {
   uint totalWordsSpoken;
   uint[string] wordCount;
}

void main() {
   // 累计剧中人物的信息
   PersonaData[string] info;
   // 填充 info
   string currentParagraph;
   foreach (line; stdin.byLine()) {
      if (line.startsWith("    ")
            && line.length > 4
            && isAlpha(line[4])) {
         // 人物在一个连续行里
         currentParagraph ~= line[3 .. $];
      } else if (line.startsWith("  ")
            && line.length > 2
```

① 即 goto，这里作者借用了 Polonius 台词里的话。——译者注

自在读书 www.zizidiary.com

```
                && isAlpha(line[2])) {
            // 人物说话
            if(!currtnt Paragraph.empty)
            addParagraph(currentParagraph, info);
            currentParagraph = to!string(line[2 .. $]);
        }
    }
    // 完成之后输出收集到信息
    printResults(info);
}
```

在了解数组是如何发挥作用之后，此段代码也就不言自明了。请把重点集中到!string(line[2..$])上面去。为什么需要它，我们还遗漏了什么吗？

foreach 循环会从标准输入设备 stdin 里读取内容，并将文本行顺序存放到变量 line 里。由于每读入一行就重新分配一个缓冲区是很浪费的操作，因此每次执行循环时 byline 都会再次利用 line 的内容。line 自身的类型为 char[] ——字符数组。

检查处理了每一行，然后丢掉它，一切都很顺畅。不过如果有代码想存储行内容，那么它最好能创建一个副本。很显然，currentParagraph 就是用于存储文本的，因此需要进行复制。于是有了 to!string，它能将任何表达式转换成一个字符串。字符串（string）类型自身是不可能被改写的，to 会负责复制为获得这个保证所需要的所有事情。

现在，如果我们不使用 to!string，这段代码依然可以编译，但结果就可能会变样，这种错漏也是很难被发现的。让程序的某一部分去修改存放于程序其他部分的数据是很难进行跟踪的，因为这是一个非局部的操作（在一个大程序里，一个人要忽略多少个 to 调用呢？）。还好它不属于这种情况，line 和 currentParagraph 的类型反映出了它们各自的功能：line 的类型为 char[]，即字符数组，可以在任何时候改写它的内容；而 currentParagraph 的类型为 string，它也是一个字符数组，但无法单个进行修改。（好奇一下：string 的完整名称为 immutable(char)[]，准确的含义是“不可变字符的连续区域”。在第 4 章会讨论字符串）。它们引用的都是同一个内存内容，因为 line 会改变 currentParagraph 的值。因此编译器会拒绝编译这种错误的代码，并需要一个副本。使用 to!string 来进行转换，大家都高兴。

另一方面，四处复制字符串时，并不需要复制底层的数据——它们可以全部引用同一块内存，因为已非常明确它们是绝不会被改写的，这也就使得字符串的复制同时拥有了安全和效率。更进一步，字符串在进行跨线程共享时也不会有任何问题，同样的原因，绝对不会有竞态出现。不变性（immutability）其实真的很强大。如果有时需要精确地修改某个单字符，则可以临时操作 char[]。

上面定义的 PersonaData 非常简单，不过结构不仅能定义数据，也可以定义其他的实体，像 private 区块、成员函数、单元测试、操作符、构造函数和析构函数。默认情况下，结构的每一个数据成员都会被初始化成自己的默认值，整数为 0，浮点数为 NaN（不是一个数字，Not a Number）[①]；数组和其他间接访问类型为 null。现在来实现 addParagraph，它会完全分割文本行，然后放置到关联数组里。

main 函数处理的行中有这么一种形式"Ham. To be, or not to be- that is the question."。我们需要从中找到第一句点“.”，以便能在具体行里分辨出人物的名字来。使用 find

① NaN 对于浮点数来说并是一个好的默认初始值，但没办法，整数没有类似这样的初始值。

自在读书 www.zizidiary.com

函数可以实现这个功能。haystack.find(needle)返回的是 haystack 中第一次出现 needle 后的右边部分（如果没有出现，find 函数会返回一个空的字符串）。此外，在收集词汇时，我们还要做一点清理工作。首先，我们必须把句子转换成小写的，这样不论单词是否大写都会被统计到同一个词汇元素上。这个调用 tolower 函数即可轻松实现。其次，我们必须消除一些干扰：标点符号算是一个，如“him.”和“him”就该统计为两个不同的单词。为实现词汇表的清理，需要向 split 函数传递一个附加参数，它的内容是一个用于消除所有杂质的正则表达式：regex("[\t,.;:?]+")。利用此参数，split 函数就会将[和]之间提及的一系列字符当做单词分隔符。也就是说，我们只需要下面这样一小段代码便可以完成很多事情：

```
void addParagraph(string line, ref PersonaData[string] info) {
    // 找出人物和句子
    line = strip(line);
    auto sentence = std.algorithm.find(line, ". ");
    if (sentence.empty) {
        return;
    }
    auto persona = line[0 .. $ - sentence.length];
    sentence = toLower(strip(sentence[2 .. $]));
    // 得到对话里的单词
    auto words = std.regex.split(sentence, regex("[ \t,.;:?]+"));
    // 插入或更新信息
    if (!(persona in info)) {
        // 此人物第一次开口说话
        info[persona] = PersonaData();
    }
    info[persona].totalWordsSpoken += words.length;
    foreach (word; words) ++info[persona].wordCount[word];
}
```

addParagraph 函数的主体是由更新关联数组的操作组成的。如果有人还未说话，则代码会往关联数组中插入一个空的、默认构造的 PersonaData 对象。因为默认构造的 uint 为零，而默认构造的关联数组为空，因此刚插入的对象已可以用于接收那些有用的信息。

最后，实现 printResults 函数来输出每一位人物的小结信息：

```
void printResults(PersonaData[string] info) {
    foreach (persona, data; info) {
        writefln("%20s %6u %6u", persona, data.totalWordsSpoken,
            data.wordCount.length);
    }
}
```

准备好测试了吗？保存并运行它吧！

```
               Queen   1104    500
                 Ros    738    338
                 For     55     45
                Fort    138    102
           Gentlemen      4      3
               Other    105     75
                Guil    349    176
                 Mar    423    231
                Capt     92     66
                Lord     70     49
                Both     44     24
                 Oph    998    401
               Ghost    683    350
```

自在读书 www.zizidiary.com

```
          All       20       17
       Player       16       14
         Laer     1507      606
          Pol     2626      870
       Priest       92       66
          Hor     2129      763
         King     4153     1251
  Cor., Volt       11       11
   Both [Mar        8        8
          Osr      379      179
         Mess      110       79
       Sailor       42       36
      Servant       11       10
    Ambassador      41       34
         Fran       64       47
        Clown      665      298
         Gent      101       77
          Ham    11901     2822
          Ber      220      135
         Volt      150      112
          Rey       80       37
```

现在变得更有意义了。毫无悬念，我们的朋友“Ham”以绝对的优势摘得了桂冠。Voltemand（Volt）的角色比较有意思：他的话不多，但即使只有那么几句话，他也尽其所能展现出了丰富的词汇量，几乎未重复过一个字。将表达多些的王后与 Ophelia 进行一下对比可以发现：王后讲的单词比 Ophelia 多出了 10%，但她多出的词汇量却不到 25%。

在输出结果里有一些干扰（如"Both [Mar"），这个对于勤奋的程序员来说很容易修复，但对那些重要的统计没什么影响。不过，修复最后这个小瑕疵也算是一种不错的指导性练习（推荐大家自行完成）。

1.6　接口和类

面向对象特性对于大型程序很重要，因此当通过小示例介绍它们时显得有点滑稽。再加上出版要求不要过度使用表现形状、动物以及雇员等内容的示例，因此我们面临着不小的难题。此外还有一件事情，那就是小示例容易掩盖多态对象创建的问题，而这一点是非常重要的。这个差点把作者给难倒了！还好现实生活中有这么一个可以用上的例子，它以问题的形式存在，而且相对较小，但还没有一个令人满意的过程解决方案。下面我们要讨论的代码改写自一个很有用的 awk 小脚本，该脚本的成长早已超出了由其设计设置的各种隐含限制。我们一起来实现一个面向对象的解决方案，它同样小巧、完整和优雅。

假设要编写一个名叫 stats 的小统计程序，它同时带有一个简单的接口：stats 可以接收统计函数作为命令行参数进行计算，它通过标准输入以空白分隔列表的形式收集要操作的数据，并在每一行输出统计结果。这里有一个会话样本：

```
$ echo 3 5 1.3 4 10 4.5 1 5 |  stats Min Max Average
1
10
4.225
$ _
```

随便哪个快速实现的脚本都可以完成这样的任务，不过随着统计函数个数的增长，实现起

自在读书 www.zizidiary.com

来就没那么快了。因此，我们需要一个更好的解决方案。我们暂时先从一些最简单的统计函数（minimum、maximum 和 average）着手，然后再想出一个扩展的设计，允许往后实现更复杂的统计函数。

有一种简单的方法，即只循环一下输入并计算出所有需要的统计值。这不是一个可扩展的设计，因为每当我们添加一个新的统计函数时，都必须对已有的代码“动手术”。如果只想要完成命令行里要求的计算，那么这个修改是很重要的。理想情形下，我们会将每一个统计函数限定在一个连续的代码块里。这样，当给程序添加新功能时，只需要简单地添加新的代码即可——正好符合“开-闭”原则[39]。

这种方法需要找出所有或者绝大部分统计函数所具有的共同点，其目的在于可以从一个地方、以一种统一的方式来操作它们。先做些说明，Min 和 Max 每次都输入一个数字，在输入完成之后立即得出一个结果。最后的结果就只有一个数字。另外，Average 必须在事后进行处理（将累计的总和除以输入的个数）。还有，每一个算法都要维护好自己的状态。当不同的计算都服从某个统一的接口，并需要保留状态时，理所当然需要让它们成为对象，并定义一个形式接口，以便控制这些对象中的任何一个或全部。

```
interface Stat {
   void accumulate(double x);
   void postprocess();
   double result();
}
```

一个接口以一组函数的形式来定义所要求的行为。当然，所有声明实现该接口的类型都必须定义它们的声明所指定的所有函数。说到实现，首先一起来看看如何定义 Min 以便符合 Stat 的要求：

```
class Min : Stat {
   private double min = double.max;
   void accumulate(double x) {
      if (x < min) {
         min = x;
      }
   }
   void postprocess() {} // 什么也不做
   double result() {
      return min;
   }
}
```

Min 是一个类——一种自定义类型，它将面向对象的很多内容引入到了 D 语言里。很明显，Min 是通过“class Min : Stat”这样的语法实现 Stat 的，并且使用了完全一样的参数和返回类型来定义 Stat 的 3 个函数（否则，编译器不会认可 Min）。Min 仅保留一个私有成员变量 min，它代表的是当前的最小值，accumulate 会将其更新。min 的初始值为最大可用数，这样第一个输入数据就会替换它。

在定义更多统计函数之前，我们先为 stats 程序编写一个测试驱动，它要读取命令行参数，创建合适的对象用于完成计算（例如，当在命令行传递 Min 时，进行最小值的计算），并通过接口 Stat 使用该对象。

```
import std.exception, std.stdio;

void main(string[] args) {
```

自在读书
www.zizidiary.com

```
    Stat[] stats;
    foreach (arg; args[1 .. $]) {
        auto newStat = cast(Stat) Object.factory("stats." ~ arg);
        enforce(newStat, "Invalid statistics function: " ~ arg);
        stats ~= newStat;
    }
    for (double x; readf(" %s ", &x) == 1; ) {
        foreach (s; stats) {
            s.accumulate(x);
        }
    }
    foreach (s; stats) {
        s.postprocess();
        writeln(s.result());
    }
}
```

此程序做了不少工作，不过也就一句话的事儿。首先，main 的声明标识与我们之前见到的有所不同——它会接收一个字符串数组。D 语言运行时支持通过命令行参数初始化该数组。第一个循环会利用 args 初始化 stats 数组。在 D 语言里，第一个参数固定为程序自己的名字（跟其他语言一样），我们利用分割 args[1 .. $]跳过第一个参数。现在来看下面这条语句：

```
auto newStat = cast(Stat) Object.factory("stats." ~ arg);
```

它有点儿长，不过套用一句情景剧台词——“我可以解释”。首先，当用做二元操作符时，~ 的作用是连接字符串，因此，如果命令行参数为 Min，则该串连接的结果就是"stats.Min"，它会传递给函数 Object.factory。Object 是所有类对象的根，它定义的静态方法 factory 会接收一个字符串，查询一个在编译期间建立的小数据库，魔法般地创建一个对象（其类型名与传入字符串一致），并返回它。如果该类不存在，Object.factory 会返回 null。为保证调用成功，所有需要做的就是在同一文件的某一个地方定义一个名叫 Min 的类。对于很多有用的应用程序来说，根据类型名来创建对象是一项很重要的功能——事实上，有一些动态语言也因此将它作为一项核心功能。语言想要拥有更多静态的用于类型化的方法，则需要依赖于运行时支持（如 D 语言或者 Java 语言），或者需要通过程序员设计一个手工注册和发现的机制。

为什么是 stats.Min 而不是只有 Min 呢？D 语言是高度模块化的，因此它没有可以让任何人放置内容的全局命名空间。每一个符号都在一个命名模块里，而且在默认情形下，模块的名称就是源文件的主名。因此，假定我们的文件叫 stats.d，那么 D 语言便会认定在该文件里定义的每一个名字都属于模块 stats。

还有一个疑点需要解决。刚获得的 Min 对象的静态类型其实并不是 Min。这听起来让人有些晕，但这是事实。你可以通过调用 Object.factory("whatever")来创建任何对象，因此返回的类型应该是可能有的所有对象类型的公共部分，即 Object。为获得新创建对象上正确的处理操作，我们必须将其转换成一个 Stat 对象，即使用众所周知的“强制类型转换”操作。在 D 语言里，表达式 cast(T) expr 会将表达式 expr 强制转换成类型 T。涉及类和接口的强制转换操作都需要进行检查，因此我们的代码相当简单。

回顾一下，我们注意到，在 main 的前 5 行里已完成了大量坚实的工作。这是最难的部分，因为代码的其余部分是水到渠成的。第二个循环每次会读取一个数（readf 负责此事），然后调用所有统计对象的 accumulate。函数 readf 会返回按照指定格式成功读入的条目的数量。

自在读书 www.zizidiary.com

在这里，格式为" %s "，它表示的是一个由任意数量空格围起来的条目（其类型由所读入元素的类型决定，在这个示例里，x 的类型为 double）。最后，程序输出所有的结果。

更多统计、继承

Max 的实现跟 Min 的实现一样简单，几乎所有内容都一样，只是在 accumulate 里有一点细微的调整。不管什么时候，当一个新的任务看起来与旧的任务差不多时，头脑里就应该浮现保持“有趣”但不“单调”的想法。重复任务就有了进行重用的机会，而能更好地挖掘出各类相似性的语言，其品质就应该是更胜一筹。需要我们解决的部分是 Min 与 Max 之间特定的相似性（希望其他统计函数也能够有）。正如我们所想的，它们似乎都属于这样一类统计函数——它们都是递增式地构建结果，并且只需要一个数字来表征结果。那就将这类统计函数叫做“递增函数”吧。

```
class IncrementalStat : Stat {
   protected double _result;
   abstract void accumulate(double x);
   void postprocess() {}
   double result() {
      return _result;
   }
}
```

抽象类（abstract class）可以看成是一个部分承诺：它实现了某些方法，但不是全部。也正因如此，导致它无法单独工作。具体化一个抽象类的方式就是继承它并完成其实现。IncrementalStat 会负责 Stat 的样板代码，但会将 accumulate 留给其派生类去实现。新的 Min 类便成为下面这个样子：

```
class Min : IncrementalStat {
   this() {
      _result = double.max;
   }
   void accumulate(double x) {
      if (x < _result) {
         _result = x;
      }
   }
}
```

类 Min 也以特殊函数 this() 的形式定义了一个构造函数，用于初始化各自的结果。与构造函数情形一致，最终代码在各种事务状态的初始化方面节省了很多工作，特别是有这么一个事实存在：其他许多统计函数（如求和、偏差、平均、标准差）都遵从类似的模式。一起来看看求平均的实现，因为它更适合更多概念的介绍：

```
class Average : IncrementalStat {
   private uint items = 0;
   this() {
      _result = 0;
   }
   void accumulate(double x) {
      _result += x;
      ++items;
   }
```

自在读书 www.zizidiary.com

```
    override void postprocess() {
        if (items) {
            _result /= items;
        }
    }
}
```

首先，Average 又引入了一个成员变量 items，通过= 0 语法（仅仅是借用了初始化的语法，其实它涉及的数量很多；还有正如在 1.5 节中讨论过的，整数类型的初始值为 0），它被初始化成 0。其次，Average 定义了一个构造函数，它会将 result 设置为 0；这是因为，与最小值或最大值有所不同，当个数为零时，平均值被定义为 0。先将 result 初始化成 NaN，然后又只是将它改写成 0，虽然这个操作看起来显得有些多此一举，但它对所有优化器来说，将所谓的“无用赋值”（dead assignment）优化掉只是唾手可得的事。最后，Average 改写了 IncrementalStat 里已定义过的 postprocess 函数。在 D 语言里，默认情况下你可以改写（继承后进行重新定义）所有类的成员函数，不过必须要指定 override，这样才能避免发生各式各样的意外（如因为排版或基类型的更改而无法进行改写，或错误地进行了改写）。如果某个成员函数被设定为 final，那么它就会禁止派生类改写该函数，这有效地阻断了动态方法的查找机制。

1.7　值与引用

先运行一个简单的实验：

```
import std.stdio;

struct MyStruct {
    int data;
}
class MyClass {
    int data;
}

void main() {
    // 测试 MyStruct 对象
    MyStruct s1;
    MyStruct s2 = s1;
    ++s2.data;
    writeln(s1.data); // 输出 0
    // 测试 MyClass 对象
    MyClass c1 = new MyClass;
    MyClass c2 = c1;
    ++c2.data;
    writeln(c1.data); // 输出 1
}
```

看起来操作 MyStruct 对象与操作 MyClass 对象似乎有很大的不同。两种情况下我们都创建了一个变量来复制另一个变量的内容，之后修改那个副本（回想一下，++是一个一元操作符，它会递增其参数）。实验表明，在复制之后，c1 和 c2 都引用了同一个底层存储内容，而相反的是，s1 和 s2 是各自独立的。

MyStruct 的行为服从“值语义”（value semantics）：每一个变量都恰好引用一个值，将一个变量赋予另一个变量时，表示的是将该变量的状态复制为另个变量的状态。复制源不会发生变化，并且两个变量会继续独立地演变。MyClass 的行为服从“引用语义”（reference semantics）：

自在读书 www.zizidiary.com

值需要显式创建（在这里是通过调用 new MyClass 来实现的），将一个类变量简单地赋予另一个变量时，表示的是两个变量都引用同一个值。

值语义容易处理，易于推理，且支持小规模数据的高效实现。而另一方面，对于某个值，在没有对它进行复制的情况下，如果不通过某些特殊方法便想去引用它，那么有些重要的程序则难以实现。值语义不适合的情形有：组成自引用类型（列表和树），像包含有其父级窗体信息的子窗体这种互斥引用结构。严格的语言都会实现某些引用语义，争辩的焦点集中在默认值。C 语言的值语义比较特殊，它支持通过指针的方式显式转成引用。C++语言除了指针之外，也定义了引用类型。有意思的是，纯的函数式语言并没有对使用引用还是使用值传语义进行区分，这是因为用户代码无法分辨出这种差异。由于纯的函数式语言不支持突变（mutation），因此你无法得知它们只是某个值的副本，还是其引用——它被冻结了，因此你无法通过更改它来验证其值是否已被共享。相反，纯的面向对象语言历来针对的都是突变密集的问题，并且明确地使用了引用语义。此外，它还进行了一些扩展——允许超大规模的灵活性，如动态调整全局系统的常量。最后，另外有一些语言则采用了将两者混合的方法，通过多级承诺支持值和引用两种类型。

D 语言系统性地采用了混合方法。定义引用类型可以使用类（class）。定义值类型或混合类型，则可以使用结构（struct）。关于它们的详细内容分别会在第 6 章和第 7 章进行描述。它们的构造函数都具备专门针对这种基本设计选择的设施。例如，结构不支持动态继承和多态性（像前面的 stats 程序里所展示的那种情况），因为这类行为与值语义是不兼容的。对象的动态多态性需要引用语义，而且试图想将其混在一起的做法都会导致很多可怕的意外。例如，在 C++语言里有一项普遍关注的危险——分割，即当不经意将一个对象当成一个值来使用时，其多态能力会突然被剥离。在 D 语言里，这种现象绝不会发生。

最后还有一个想法就是结构理所当然就成为实现更灵活设计的选择。通过定义结构，你可以表现出任何想要的语义，如热复制（eager-copy）值、惰性复制（lazy copying）或者介于两者之间的任何内容。惰性复制有时也叫写时复制（copy-on-write）或引用计数（reference counting）。你也可以通过使用类对象或者结构对象里的指针来定义引用语义。在另一方面，有一些操作可能要求能对一些相当高级的技术进行理解，而使用类提供的简单性和统一性则可以顺利跨越这个障碍。

1.8　小结

由于本章是介绍性的，因此有一些概念和示例掩盖了某些细节，并且假设大家熟悉其他部分。当然，有经验的程序员可以轻易地找到方法来完成和改进这些示例。

不考虑特殊情况，本章包含的内容适合于所有人。如果你属于实践型和一丝不苟的代码编写者，可能已用赏识的眼光看到了数组和关联数组的简洁性。在各类大小项目中，这两个概念都各自改变了日复一日的代码简化操作。如果你喜欢面向对象，那么接口和类无疑很适合你，同时建议将 D 语言所具有的良好向上的扩展性应用到大型项目中。如果需要将 D 语言用于简短的脚本中，那么本章也展示了一些操作文件的简短脚本，它们都很容易编写和运行。

通常，概述的内容都比较长。不过，关注基础内容，并且确保简单的事情依然简单是很有意义的。

自在读书 www.zizidiary.com

第2章

基本类型、表达式

如果你曾用C、C++、Java或C#编写过程序，那么在理解D语言的基本类型和表达式时将会有一种轻车熟路的感觉——还可能觉得改进不少。在许多编程任务里，操作基本类型的值算得上是家常便饭，语言提供的与个人喜好存在交互的内容会伴随你很久很久，让你的生活充满欢乐或痛苦。世上没有完美的方法，许多迫切想要得到的方法往往都充满矛盾，还会受到主观因素的影响。因此，对于一门语言，想要找到一种让所有人都满意的解决方案是不太可能的。类型系统太严格会将责任置于错误的地方，因为就连最简单的惯用语法，程序员也必须要与编译器纠缠半天才能使其接受它；太宽松的话，又会让程序员陷入可验证性、效率或二者兼有的麻烦中。

在静态类型的编译型语言家族里，D语言的基本类型系统在其成员资格所界定的范围内创造了多个小奇迹。类型推断（type inference）、值域传播（value range propagation）、各类操作符重载决断以及精心设计的类型自动转换网，它们联合一起让D语言的类型系统成为一个考虑周全的辅助工具，通常它只会在真正有必要的时候，才会引发讨论和关注。

这些基本类型可以被划分成如下几类。

- 无值的类型：`void`，适合它的情形为形式上需要一个类型，但不会产生任何有意义的值。
- 布尔类型：`bool`，它有两种可能值，即`true`（真）和`false`（假）。
- 整数类型：`byte`、`short`、`int`和`long`，以及它们所对应的无符号类型`ubyte`、`ushort`、`uint`和`ulong`。
- 真正的浮点类型：`float`、`double`和`real`。
- 字符类型：`char`、`wchar`和`dchar`，它们都是数字，只不过语言会将它们理解为Unicode字符串编码。

表2-1简要描述了各类基本类型，其中包含了它们的大小以及默认初始值。在D语言里，如果只是定义而没进行初始化，那么所有变量都会被默认值初始化。默认值的访问方式为`<类型>.init`，例如，`int.init`即为0。

自在读书
www.zizidiary.com

表 2-1　D 语言基本类型

名称	描述	默认初始值（<类型>.init）
void	无值	n/a
bool	布尔值	false
byte	有符号 8 位	0
ubyte	无符号 8 位	0
short	有符号 16 位	0
ushort	无符号 16 位	0
int	有符号 32 位	0
uint	无符号 32 位	0
long	有符号 64 位	0
ulong	无符号 64 位	0
float	32 位浮点数	float.nan
double	64 位浮点数	double.nan
real	硬件最大值	real.nan
char	无符号 8 位，UTF-8	0xFF
wchar	无符号 16 位，UTF-16	0xFFFF
dchar	无符号 32 位，UTF-32	0x0000FFFF

2.1　符号

符号（symbol）指的是一个大小写敏感的字符串，它由字母或下划线开头，后面紧跟任意数量的字母、下划线或数字。此规则的唯一例外是以两个下划线开头的符号，它们已被 D 语言的具体实现所预留。符号的开头只允许有一个下划线，它们常被用作成员变量。

关于 D 语言符号有一个细节很有意思，即对国际化的支持：在上面的定义中，字母不仅仅表示了 A 到 Z 以及 a 到 z 这样一些罗马字母表里的字符，同时还表示了 C99 标准[33]里的统一字符。

例如，abc、α5、_、Γ_1、_AbC、Ab9C 以及_9x 都是有效的符号，但 9abc、__以及__abc 都不是。

如果符号的前缀是一个点，如.likeThis，那么便会在模块作用域里去查找它，而非在当前的词法嵌套域里查找。作为前缀的点操作符拥有与普通符号一样的优先级。

特殊符号

表 2-2 里列出的符号是 D 语言预留的关键字，任何情形下用户代码都不能定义它们。

表 2-2　D 语言关键字

abstract	•	long	static
alias	else	•	struct
align	enum	macro	super
asm	export	mixin	switch
assert	extern	module	synchronized

自在读书 www.zizidiary.com

续表

auto	•	•	•
•	false	new	template
body	final	nothrow	this
bool	finally	null	throw
break	float	•	true
byte	for	out	try
•	foreach	override	typeid
case	function	•	typeof
cast	•	package	•
catch	goto	pragma	ubyte
char	•	private	uint
class	if	protected	ulong
const	immutable	public	union
continue	import	pure	unittest
•	in	•	ushort
dchar	inout	real	•
debug	int	ref	version
default	interface	return	void
delegate	invariant	•	•
deprecated	is	scope	wchar
do	•	shared	while
double	lazy	short	with

有些符号被认定为基本表达式。在方法定义的内部，特殊符号 this 指代的是当前对象，super 限定为静态和动态查找当前对象的基对象，这一点在第 6 章会进行讨论。符号$只在索引表达式或分割表达式里有效，它会计算出正被索引的数组的长度。符号 null 指代的是一个空的对象、数组或指针。

基本表达式 typeid(T) 返回的是关于类型 T 的信息（更多信息请查阅具体编译器的实现文档）。

2.2　字面量

2.2.1　布尔字面量

布尔字面量（Boolean literal）有 true 和 false。

2.2.2　整数字面量

D 语言支持十进制、十六进制以及二进制整数字面量。十进制常量就是一串数字，其后缀允许是 L、U、u、LU、Lu、UL 或者 uL 等内容。十进制字面量的类型推断规则如下。

- *无后缀*：其类型为 int 与 long 中能优先容纳下该值的那一个。
- *只有 U/u*：其类型为 uint 与 ulong 中能优先容纳下该值的那一个。

自在读书 www.zizidiary.com

- 只有 L：其类型为 long。
- U/u 和 L 都存在：其类型为 ulong。

例如：

```
auto
    a = 42,             // a 的类型为 int
    b = 42u,            // b 的类型为 uint
    c = 42UL,           // c 的类型为 ulong
    d = 4000000000,     // long：无法放入 int
    e = 4000000000u,    // uint：它能放入 uint
    f = 5000000000u;    // ulong：无法放入 uint
```

在一个数里可以随意插入下划线（只是不要插入到第一位，以免最终创建成一个标识符）。下划线有助于清晰地书写很大的数：

```
auto targetSalary = 15_000_000;
```

十六进制整数字面量的书写方法是前缀 0x 或 0X 之后紧跟 0～9、a～f、A～F 或者_构成的字母串。首位为 0，后面紧跟 0～7 构成的串或空串，组成的是八进制字面量。最后，构建二进制字面量的方法是在 0b 或 0B 后面紧跟由 0、1 和下划线构成的串。所有这些字面量与十进制常量类似，都可以带上后缀。

图 2-1 胜过千言万语，它简洁明了地定义了整数字面量的语法。查看这个自动机的规则是：（1）每条边会将与之对应的标签当作输入；（2）自动机会尽可能多地接收输入[①]。当停留在终点状态（即双重圆圈）时，即表明一个数字解析成功。

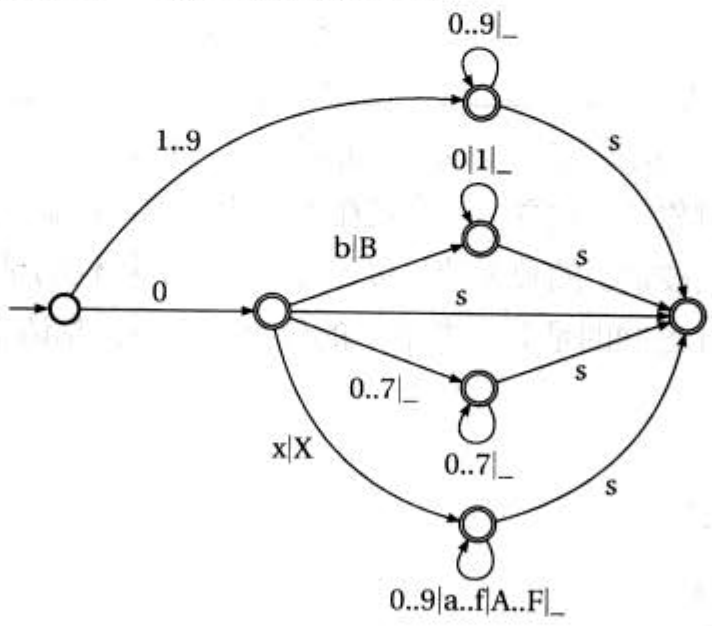


图 2-1　理解 D 语言的整数字面量。本自动机会尝试一步一步连续地执行下去（同时处理掉与所遇边相对应的输入），直到其必须结束为止。当停留在终点状态（双重圆圈）时，即表明一个数字解析成功。*s* 代表的是后缀，取值可以是：U|u|L|UL|uL|Lu|LU

2.2.3　浮点数字面量

浮点数字面量可以是十进制或十六进制。在仅仅是定义整数字面量方面，十进制浮点数字面量比较容易定义：一个十进制浮点数字面量由一个十进制数字面量构成，它也可以在任何位置包含一

① 从理论上来讲，图 2-1 和图 2-2 都是“确定有限自动机”（DFA）。

自在读书 www.zizidiary.com

个点“.”，后面可以紧跟一个指数和（或）后缀。指数为 e、E、e+、E+、e 或 E-之后，后面再紧跟一个无符号十进制数字面量。后缀可以是 f、F 或者 L。显然，在“.”、“e/E”和“f/F”之间至少需要有一项存在，以防止浮点数在字面上没了小数点而成为一个整数字面量。后缀 f/F 如果存在，则字面量的类型会强制为 float；而 L 会将其强制为 real；否则，字面量的类型为 double。

十六进制浮点常量看起来可能有些奇怪，但它们在准确书写常量时却非常有用。在内部，浮点值都是以基数为 2 的形式存储的，因此一个实数以基数 10 的形式显示时，它其实是进行了一个基数的转换；而由于 10 并不是 2 的次方，因此该转换只能是近似的。与此相反，十六进制的记数法让你能够记录下恰好与它们表现出的值一致的浮点数。关于浮点数是如何存储的完整论述已超出本书的范围，不过可以保证 D 语言的实现都是使用的 IEEE 754 格式，更多详尽的参数资料请上网搜索（如查找“IEEE 754 浮点数格式”）。

一个十六进制浮点常量构成是：前缀 0x 或 0X 之后紧跟一串可在任何位置包含一个小数点的十六进制数字。对于指数，它是由 p、P、p+、P+、p-或 P-当中的任何一个做引导，后面紧跟多个十进制（可不是十六进制哦！）数字。只有所谓的尾数（在指数之前的小数部分）才是以十六进制形式进行表达的；指数自己是一个十进制整数。十六进制浮点常量的指数在最终的数里是表示成 2 的指数（而非像十进制里的情况表示成 10 的指数）。最后，还可选择后缀 f、F 或 L 来补齐该常量[①]。一起来看看相关的示例。

```
auto
    a = 1.0,                          // a 的类型为 double
    b = .345E2f,                      // b 为 34.5，其类型为 float
    c = 10f,                          // 因为后缀，所以 c 的类型为 float
    d = 10.,                          // d 的类型为 double
    e = 0x1.fffffffffffffp1023,       // e 为最大的 double 类型值
    f = 0XFp1F;                       // f 为 30.0，其类型为 float
```

图 2-2 简明描述了 D 语言的浮点数字面量。执行该自动机的规则与整数常量自动机的一样：在字面量里读取字符时会进行变换，而且尝试的是最长路径。这里展示的自动机可以分辨出好几种不常见的点，例如 0x.p1 和 0xp1 尽管有些奇怪，但都是 0 的合法表示方法，但像 0e1、.e1 和 0x0.0 这样结构都是不允许的。

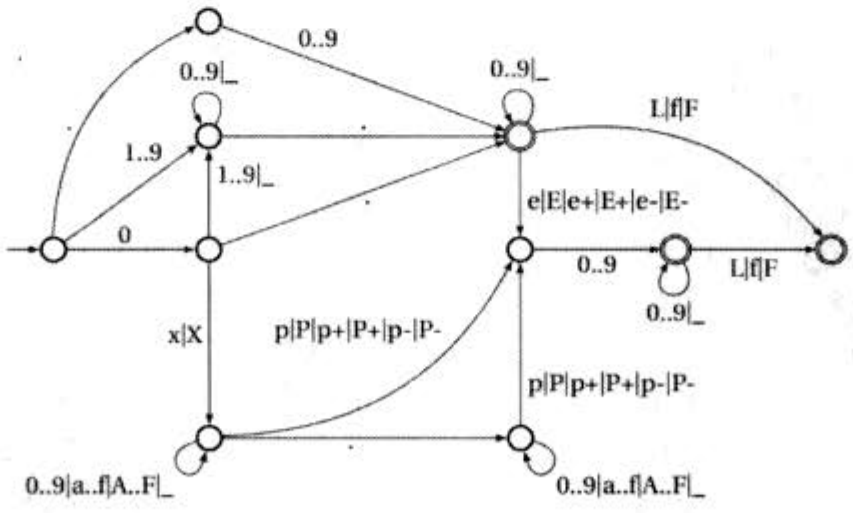


图 2-2　理解浮点数字面量

① 不错，该语法是有些奇怪，不过 D 语言只是复制了一 C99 的语法，并没有自己特意去发明一种新的记号。

自在读书 www.zizidiary.com

2.2.4　字符字面量

字符字面量指的是使用单引号引起来的单个字符，如'a'。实际的引号符号必须使用反斜线进行转义，如'\"'。实际上，与其他语言一样，D 语言也定义了好几个转义序列，在表 2-3 里全部列出来。除了包含标准控制字符外，D 语言还定义了组成 Unicode 字符的方法，即通过使用'\u03C9'（\u 后面紧跟 4 个十六进制数字）、'\U0000211C'（\U 后面紧跟 8 个十六进制数字）或'\©'（以\&作为开头，并以;作为结尾的命名实体）这样的记号来实现。第一个是 ω 的 Unicode 码，第二个是 R 的美观写法 $\mathscr{R}$，而最后一个是让人惧怕的版权符号©。需要一个完整列表时，可以上网搜索“Unicode 表”。

2.2.5　字符串字面量

在了解了如何表示字符之后，再了解字符串字面量就不会有什么问题。D 语言在操作字符串方面相当厉害，其部分原因就在于强大的字符串字面量表示方式。与其他字符串操作友好的语言一样，D 语言也对引号字符串（在其内部可以应用表 2-3 里列出的转义序列）和所见即所得（What You See Is What You Get，WYSIWYG）字符串（编译器只是进行简单地解析，并不会去解读转义序列）进行了区别对待。所见即所得风格对于表示要求包含很多的转义符号的字符串相当有用，这里有两个臭名昭著的例子，它们分别是正则表达式和 Windows 路径名称。

表 2-3　D 语言转义序列

序列	类型	描述
\'	char	单引号（存在歧义时）
\"	char	双引号（存在歧义时）
\\	char	反斜线
\a	char	响铃（ASCII 7）
\b	char	退格（ASCII 10）
\f	char	换页（ASCII 14）
\n	char	换行（ASCII 12）
\r	char	回车（ASCII 15）
\t	char	水平制表（ASCII 9）
\v	char	垂直制表（ASCII 11）
\‹1 到 3 个八进制数字›	char	八进制形式的 UTF-8 字符（必须≤377_8）
\x‹2 个十六进制数字›	char	十六进制形式的 UTF-8 字符
\u‹4 个十六进制数字›	wchar	十六进制形式的 UTF-16 字符
\U‹8 个十六进制数字›	dchar	十六进制形式的 UTF-32 字符
\&‹命名字符实体›;	dchar	符号化的 Unicode 字符

引号字符串指的是由双引号引起来的字符序列，如"like this"。所有列在表 2-3 里的转义序列在引号字符串内都具有意义。

自在读书 www.zizidiary.com

当各种类型的字符串并排在一起时会被自动拼接：

```
auto crlf = "\r\n";
auto a = "This is a string with \"quotes\" in it, and also
a newline, actually two" "\n";
```

在上面的代码里，also 之后的换行是刻意的：字符串字面量可以嵌入换行符（源代码里实际的换行符，而非反斜线之后跟一个 n），并照原样进行存储。

2.2.5.1　所见即所得、十六进制以及导入的字符串字面量

所见即所得字符串可以用 r 作为开始，以“”作为结尾（如 r"like this"），也可以在开始和结尾部分都使用反引号（如'like that'）。所有字符（除了各自的终止符以外）都可以出现在所见即所得字符串里，并会以表面值形式进行存储。即是说，无法在双引号所见即所得字符串里表示双引号本身。这不是什么大问题，因为可以使用各种语法来拼接字面量字符串。例如：

```
auto a =  r"String with a \ and a " `"` " inside.";
```

出于实用的目的，也可以考虑通过使用序列<"`"`">来对处于<r"string">里的双引号进行编码，以及将'string'里的反引号编码成像<`"`"`>这个样子。祝大家的引号数得开心。

D 语言还定义了第三种字面量字符串：十六进制字符串。它是一个由十六进制数字和空白符（会被忽略掉）组成的字符串，界定符有“x""和“""。在定义原始数据时十六进制字符串比较有用，除了十六进制数字以外，编译器并不会将其中的内容解释成 Unicode 字符或其他什么东西。在十六进制字符串内部的空格会被忽略掉。

```
auto
   a = x"0A",            // 等同于"\x0A"
   b = x"00 F BCD 32"; // 等同于"\x00\xFB\xCD\x32"
```

如果此时在你的“黑客”头脑里已浮现出了将二进制资源嵌入 D 语言程序的想法，那么当你听说还有一种更强大的定义字符串的方式时，一定会很高兴。这种方式就是通过文件来定义！

```
auto x = import("resource.bin");
```

在编译期间，x 会被初始化成文件 resource.bin 的实际内容(这一点与 C 语言的#include 方法不同，上面的操作是将文件包含为数据，而非代码)。出于安全的考虑，它只认可相对路径和通过编译器开关控制的搜索路径。参考实现 dmd 是利用-J 标志来控制字符串包含路径的。

源自 import 操作的字符串不会要求检查 UTF-8 正确性。它是为支持导入二进制资源而故意做的设计。

2.2.5.2　字面量字符串的类型

字面量字符串的类型是什么呢？先来运行一个简单的测试程序：

```
import std.stdio;
void main() {
   writeln(typeid(typeof("Hello, world!")));
}
```

内建的操作符 typeof 能获得一个表达式的类型，而 typeid 能将该类型变成一个可输出的字符串。这个小程序的输出为：

自在读书 www.zizidiary.com

```
immutable(char)[]
```

它表明字面量字符串是一个由不可变字符构成的数组。其实，在这段示例代码里使用的类型 string 是一个较长类型 immutable(char)[]的简记符号。下面从三个方面深入细节地看一下字符串字面量类型：不变性、长度和基字符类型。

不变性。字面量字符串存放于“只读”内存。它的意思并不是说它们存储在不可删除的内存芯片里或者存储在被操作系统保护的内存里，而是说语言保证不会去改写这个字符串的内存。关键字 immutable 体现了这种保证，在编译期间，它不允许有任何可能会修改不可变数据内容的操作：

```
auto a = "Nobody can change me";
a[0] = 'X'; // 出错！不能修改 immutable 字符串！
```

关键字 immutable 是一个类型修饰符（第 8 章会讨论修饰符），它可对放置于其右边的任何类型进行操作，同时遵从括号规则。假设有 immutable(char)[] str，那么在 str 里的每一个字符都是不可变的，但 str 却可以引用不同的字符串：

```
immutable(char)[] str = "One";
str[0] = 'X';                    // 出错！不能对 immutable(char)进行赋值！
str = "Two";                     // 正确，对 str 进行了重新绑定
```

另一方面，如果不存在括号，那么 immutable 会修饰整个数组：

```
immutable char[] str = "One";
str[0] = 'X';                    // 出错！
str = "Two";                     // 出错！
```

不变性有很多优点，换言之，即 immutable 为允许任意跨模块和线程数据共享（详见第 13 章）提供了很多保证。由于字符串的字符是不可更改的，因此绝不会出现竞态（contention），从而保证共享是安全和有效的。

长度。在编译期间，字面量字符串的长度（如"Hello, world!"的长度为 13）很显然是已知的。给每一个字符串指定一个最精确的类型似乎也很自然，例如"Hello, world!"就可以标识为类型 char[13]，即一个恰好拥有 13 个字符的数组。但是，Pascal 语言的应用经验表明，静态大小的数组使用起来非常不方便。因此，在 D 语言里字面量的类型没有包含长度信息。不过，如果真的想要固定大小的字符串，也可以通过显式指定长度来创建这样的字符串：

```
immutable(char)[13] a = "Hello, world!";
char[13] b = "Hello, world!";
```

对任意类型 T，大小固定的数组类型 T[N]可隐式转换到动态大小的类型为 T[]的数组。在此过程中不会丢失任何信息，这是因为动态大小的数组也记住了它们的长度：

```
import std.stdio;

void main() {
    immutable(char)[3] a = "Hi!";
    immutable(char)[] b = a;
    writeln(a.length, " ", b.length); // 输出"3 3"
}
```

基字符类型。最后一点，字符串字面量允许将 char、wchar 或 dchar 作为它们的基字符类型。当然，没有必要使用冗长的类型名称 immutable(char)[]、immutable(wchar)[]和 immutable(dchar)[]，它们都分别对应有方便使用的别名，即 string、wstring 和

自在读书 www.zizidiary.com

dstring。如果字面量字符串至少包含了一个 4 字节的 dchar 字符，那么它的类型即为 dstring；否则，如果该字符串至少包含了一个 2 字节的 wchar 字符，则它的类型即为 wstring；否则，该字符串的类型为 string。如果字符串期望的类型与推断的类型不一致，那么字面量也会像示例里演示的那样安静地接受：

```
wstring x = "Hello, wide world!";         // UTF-16 编码
dstring y = "Hello, even wider world!"; // UTF-32 编码
```

若想要改写字符串推断结果，则可以在字符串字面量后面添加后缀 c、w 或 d（类似于字符字面量的后缀），从而分别将其类型强制为 string、wstring 或 dstring。

2.2.6　数组与关联数组字面量

字符串属于特殊的数组，它们具有自己的字面语法。现在该如何表示其他类型（如 int 或 double）的数组字面量呢？数组字面量可表示成一个使用方括号括起来的、以逗号分隔的值序列：

```
auto somePrimes = [ 2u, 3, 5, 7, 11, 13 ];
auto someDoubles = [ 1.5, 3, 4.5 ];
```

数组的大小可以根据按逗号进行分隔的列表的长度计算出来。与字符串字面量有所不同，数组字面量是可变的，因此在初始化之后可以对它们进行更改：

```
auto constants = [ 2.71, 3.14, 6.023e22 ];
constants[0] = 2.21953167;  // "moving sofa"常量值
auto salutations = [ "hi", "hello", "yo" ];
salutations[2] = "Ave Caesar";
```

请注意，可以对 salutation 里某个位置上的元素进行重新分配，但不能更换存储在该位置上的字符串的内容。这正好是期望的结果，因为数组中的成员并不会影响对字符串所能执行的操作。

数组元素的类型是由数组中的所有元素共同决定的，其计算方式是通过条件操作符“?:”（在 2.3.16 节会讲到）来完成的。对于拥有多个元素的字面量 lit，编译器会应用表达式 true ? lit[0] : lit[1]，并将该表达式的类型存储为类型 L。然后，在 lit 中，对于从 lit[2] 直到最后一个元素里的每一个元素，编译器都会计算 true ? L.init : lit[i]的类型，并将该类型存回到 L 里。最后的 L 即为数组元素的类型。

与真实的情况相比，这个听起来复杂了很多。简单来说，就是数组元素的类型是通过波兰式的民主共识建立起来的——找到一个类型，让所有元素都能隐式地转换到它。例如，[1, 2, 2.2]的类型为 double，而[1, 2, 3u]的类型为 uint，原因就是 int 和 uint 在执行?:运算时，结果为 uint。

关联数组字面量的定义语法如下：

```
auto famousNamedConstants =
  [ "pi" : 3.14, "e" : 2.71, "moving sofa" : 2.22 ];
```

在关联数组字面量里的每一个槽都拥有“键 : 值”这样的形式。关联数组字面量的键类型计算方法为，在概念上所有键放置到一个数组，然后使用上面讨论过的方法来计算该数组的类

自在读书 www.zizidiary.com

型。值类型的计算方法与之非常相似。一旦键类型 K 和值类型 V 都被计算出来，则该字面量的类型即为 V[K]。例如，famousNamedConstants 的类型为 double[string]。

2.2.7　函数字面量

在有些语言里，每一个函数的名字都是在其定义的位置选择的。后期对该函数的调用都使用它的名字。另外有一些语言允许在使用它们的那个地方定义匿名函数（即众所周知的 λ 函数）。这种特性在强大的高阶函数（即用作参数并/或返回其他函数的函数）惯用法（idom）[①]方面非常有用。只要是希望提供函数名的地方，D 语言的函数字面量都支持就地定义匿名函数。

本章唯一关注的重点在于要展示如何定义函数字面量，同时还会展示一些有意义的用例。关于这一不凡特性所具有的强大用途的说明，在第 5 章再进行详细讲解，请忍耐一下。下面是函数字面量的基本语法：

```
auto f = function double(int x) { return x / 10.; };
auto a = f(5);
assert(a == 0.5);
```

函数字面量的定义遵从常规函数定义的语法，唯一不同之处在于，在定义的前面需要添加关键字 function，而不需要名称。上面的代码并没有过多使用匿名，因为匿名函数会立即绑定到符号 f。符号 f 的类型为“指向函数的指针，该函数会接收一个 int 并返回一个 double”。该类型自身被拼写为 double function(int)（请注意，关键字 function 被换到了返回类型的后面），因此 f 的等效定义为：

```
double function(int) f = function double(int x) { return x / 10.; };
```

function 与 double 之间的交换看起来有些奇怪，事实上它让大家都变得非常好过，这是因为它把函数字面量与其类型进行了区分。这样的内容是很容易记住的：在字面量里，function 先出现；而在函数的类型里，function 代替了函数的名称。

为简化函数字面量的定义，你也可以省略返回类型，而编译器会帮你推断它，这是因为它直接提供了函数体：

```
auto f = function(int x) { return x / 10.; };
```

上面的函数字面量只使用了自己的参数 x，因此其含义只通过查看函数字面量的函数体便可以判断出来，而无需依赖使用它的环境。不过，如果函数字面量需要使用存在于调用点的数据，而且该数据不是通过参数形式传递的，那该怎么办呢？此时就必须将 function 替换成 delegate：

```
int c = 2;
auto f = delegate double(int x) { return c * x / 10.; };
auto a = f(5);
assert(a == 1);
c = 3;
auto b = f(5);
assert(b == 1.5);
```

① 就软件开发阶段而言，模式可分为分析模式（andysis pattern）、架构模式（architeeture pattern）、设试模式（dseign pattern）和惯用法（idiom）。其中，惯用法指的是某种编程语言所特有的低级别模式，如 C++语言所支持的多继承表达方法，并不是所有语言都支持的。——译者注

自在读书 www.zizidiary.com

F 的类型现在变成了 double delegate(int)。所有针对 function（函数）的类型推断都可以不加更改地应用到 delegate（委托）。这时大家是否会问：既然 delegate 能完成所有 function 做的工作(当然,delegate 也可以使用所在环境里的变量,但这不是它的主要负责内容),那为什么一开始还要为 function 费心呢？难道我们就不能只使用 delegate 吗？答案很简单：效率。很明显 delegate 会访问更多的信息，因此按照某些不可抗拒的自然法则，它必须为此付出代价。其实，function 的大小就是指针的大小，而 delegate 的大小是它的两倍（一个指针用于函数，另一个则用于环境)。

2.3　操作符

接下来的几个小节将以优先级递减的方式详细描述 D 语言的操作符。这个顺序也对应于在一个更大块里进行分组和对小的子表达式进行计算的自然顺序。

有两组与操作符紧密相关的概念：左值（lvalue）对右值（rvalue），以及数值转换规则。下面两小节介绍必要的定义。

2.3.1　左值和右值

许多操作符只在它们左边满足特定条件的时候才能工作。例如，不需要复杂的理由就能辨别出来像 5 = 10 这样的赋值肯定是无效的。想要成功赋值，左边的操作数必须是一个左值。现在来准确定义一下左值（顺带右值一起，它们是互补的)。从历史的角度看，这两个术语实际源于像 a = b 这种赋值表达式里的值的位置：a 处于左边，因此它就是左值；而 b 处在右边，因此它是一个右值。

通过纯枚举的方式进行定义，左值包含的内容有：

- 所有变量，包括函数参数，即使它们无法被有效修改（如使用 immutable 进行修饰)；
- 数组和关联数组的元素；
- 结构和类的域（在后面讨论)；
- 函数返回的 ref 类型（在后面讨论)。

所有左值都可以当成右值使用。而右值还包含了上面未明确提及的内容，如字面量、枚举值（由 enum 类型引用，请参考 7.3 节）以及像 x + 5 这类表达式的结果。请注意，成为左值是允许赋值的必要但不充分条件：还必须满足其他的几项语义检查，如访问权限（参考第 6 章）和互斥权限（参考第 8 章)。

2.3.2　隐式数值转换

我们已初步接触过隐式转换的话题，现在来做个全面解析。只要涉及数值转换，就要记住下面几条简单的规则。

自在读书 www.zizidiary.com

（1）如果一个表达式在 C 语言里能进行编译，则在 D 语言里也能被编译。其类型在这两种语言里是一样的（注意：并不是所有 C 语言的表达式都一定会被 D 语言所接受）。

（2）整数值不会隐式转换成一个更窄的值。

（3）浮点值不会隐式转换成整数值。

（4）所有数值（整数或浮点数）都可以隐式地转换成任何浮点值。

规则 1 让事情比不这样做的情况稍显复杂些，不过为了能让大家轻松地将整个函数复制、粘贴到 D 语言程序里，D 语言尽量与 C 和 C++语言的相关内容保持了重叠。如果出于安全或可移植性的原因，D 语言突然拒绝编译某些部分也是很正常的。不过如果在编译了 2000 行的加密包之后，运行结果却不相同，那么对于遇到这种倒霉事儿的人来说生活就没那么轻松了。不过，规则 2 相比 C 和 C++语言加强了很多。因此当进行代码移植时，间歇性的诊断会指出代码不够精细的地方，并提示你插入恰当的检查和显式的强制类型转换。

图 2-3 展示了全部数值类型的转换规则。在转换过程中，会选择最短路径。当两条路径拥有相同的长度时，转换的结果是相同的。不管转换经历的步数有多长，转换过程也只会当成一步来转换，并且在这些转换当中不存在优先级或者顺序——要么一个类型能转换成另一个，要么不能转。

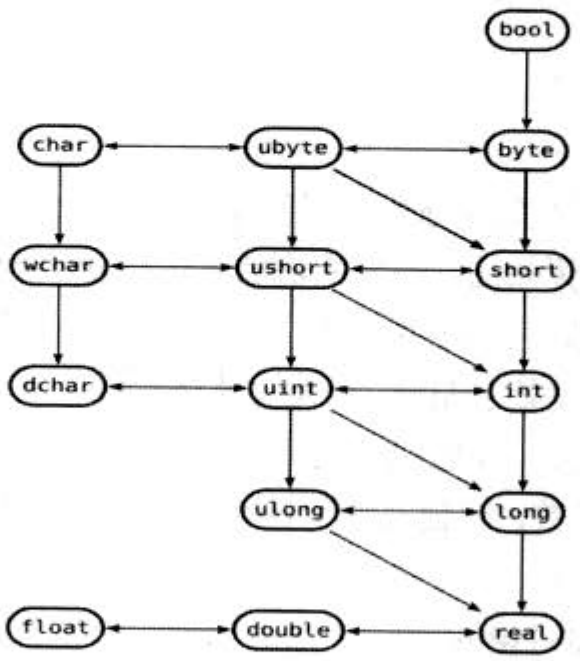


图 2-3　隐式数值转换。只要在此图中从源类型到目标类型存在一条有向路径，那么该源类型便能够自动转换成目标类型。转换时会选择最短路径，并且不管实际路径有多长也只会当成一步来转换。如果值域传播经验证是有效的，则反方向的转换也是可以的。

值域传播

按照上面描述的规则，一个普通的数字（如 42），可以明确地认为是 int 类型的。现在来看看下面同样普通的初始化操作：

```
ubyte x = 42;
```

遵从类型检查的必然规律，42 首先会被识别为 int。该 int 然后又赋给了 x，这一过程引发一个强制转换。允许这种无条件的强制转换是相当危险的（存在大量的 int 实际上都无法容纳在 ubyte 里面）。另一方面，对明显正确的代码要求进行类型强制转换是完全无法让人感到满意的。

自在读书 www.zizidiary.com

D 语言以一种智能的方法破解了这一难题，这个方法的灵感源自名叫“值域传播”（value range propagation）的编译器优化技术，即在表达式里的每一个值都有一个由最小可能值和最大可能值构成的范围与之关联。在编译期间会跟踪这些界限。当某个值被赋给一个更窄的类型时，只有值的范围适合目标类型，编译器才会认可该赋值操作。对于 42 这样的常量，很显然最小可能值和最大可能值都是 42，因此赋值操作进行得很顺利。

当然，普通的情况处理起来要容易一些，但是值域传播还可以在更多有意义的环境里进行正确性检查。假设有一个函数，它可以从一个 int 类型数里提取出最低有效字节位和最高有效字节：

```
void fun(int val) {
   ubyte lsByte = val & 0xFF;
   ubyte hsByte = val >>> 24;
   ...
}
```

不管输入值 val 是什么，这段代码都是正确的。第一个表达式是对值进行掩码操作以清除所有的高位，而第二个表达式进行的是移位操作，目的是让 val 的最高有效位移到最低有效位上，而其他部分为 0。

实际上，编译器之所以能正确地输入 fun 是因为：在第一种情况它会计算 val & 0xFF 的范围，不管 val 的值是多少，其结果都在 0～255；而在第二种情况，它会计算 val >>> 24 的范围，结果还是在 0～255。如果试验一下结果值不一定能容纳于 ubyte 类型（如 val & 0x1FF 或 val >>> 23）的操作，那么编译器就不会接受这样的代码。

值域传播能“理解”所有的算术和逻辑。例如，一个 uint 除以 100 000 以后总是能容纳在 ushort 里，并且在复杂表达式（比如在掩码操作之后再执行除法）里也一样。例如：

```
void fun(int val) {
   ubyte x = (val & 0xF0F0) / 300;
   ...
}
```

在上面的示例里，操作符“&”会将界限设置成 0～0xF0F0（十进制为 61 680），而除法则会将界限设置为 0～205。在该范围之内的任何数字都能容纳于 ubyte。

使用值域传播来辨别窄转换的正确与否是一种不完美且保守的机制。主要原因在于，值域跟踪的范围还是局限在一个表达式内部，而没有涉及多个连续的表达式。例如：

```
void fun(int x) {
   if (x >= 0 && x < 42) {
      ubyte y = x;  // 出错！不能将 int 强制转换为 ubyte！
      ...
   }
}
```

显然，这个初始化是正确的，但编译器无法识别它。它有可能会使编译器的实现变得更加复杂，而且大大降低编译速度。因此，编译器这才决定降低每一个表达式的值域传播的灵敏度。在实现此特性之后的经验表明：这种保守的估计能够消除程序里出现的绝大部分不恰当的强制类型转换错误。最后对于那些属于误判的错误，可以使用 cast 表达式（请参考 2.3.6.7 节）来解决。

自在读书 www.zizidiary.com

2.3.3　数值操作符的类型

下面的几个小节都介绍能应用于数字类型的操作符。对于数值操作符所产生的值，其类型是通过几条新规则推算出来的。在能被想到的规则中，它们算不上是最好的，但却非常简单、一致和有系统性。

一元操作符所产生的类型与操作数类型一样，但取反操作符“!”（在 2.3.6.6 节有定义）例外，它总会产生 bool 类型。对于二元操作符，结果类型的计算遵从下面的规则：

- 如果在操作符的应用中至少有一个是浮点类型，那么其结构即为所涉及的最大浮点类型。
- 否则，如果其中至少一个是 ulong 类型，而其他的类型在应用之前能够隐式转换为 ulong，则结果类型为 ulong。
- 否则，如果其中至少一个是 long 类型，而其他的类型在应用之前能够隐式转换为 long，则结果类型为 long。
- 否则，如果其中至少一个是 uint 类型，而其他的类型在应用之前能够隐式转换为 uint，则结果类型为 uint。
- 否则，将两个操作数隐式转换成 int 并应用操作符。结果类型为 int。

所有隐式转换都会选择图 2-3 所描绘的最短路径。这个细节很重要，请看下面的内容：

```
ushort x = 60_000;
assert(x / 10 == 6000);
```

在除法运算里，10 的类型为 int，而根据上面的规则，x 在执行操作之前会隐式转换成 int 类型。图 2-3 显示了好几条可能的路径，其中有直接的转换 ushort→int 以及稍微长一点（有一个跳转）的转换 ushort→short→int。第二条不符合要求，因为将 60000 转换成 short 会生成-5536，而它进一步提升至 int 会导致 assert 失败。选择转换图里的最短路径能保证存储值更准确。

2.3.4　基本表达式

基本表达式是进行计算的原子操作。我们已见识过了符号（请参考 2.1 节）、布尔字面量 true 和 false（请参考 2.2.1 节）、整数字面量（请参考 2.2.2 节）、浮点数字面量（请参考 2.2.3 节）、字符字面量（请参考 2.2.4 节）、字符串字面量（请参考 2.2.5 节）、数组字面量（请参考 2.2.6 节）以及函数字面量（请参考 2.2.7 节）。跟字面量 null 一样，它们都是基本表达式。下面各小节描述的是其他一些基本子表达式，其中包括 assert 表达式、mixin 表达式、is 表达式和括号表达式。

2.3.4.1　assert 表达式

有很多表达式和语句，包括 assert（断言）自身，都使用了“非零值”（nonzero）的概念。这些值可以为：（a）数字或字符类型，此时，非零值是有确切含义的，（b）布尔类型（非零值表示的是 true），或者（c）数组、引用以及指针类型（非零值表示的是非 null）。

自在读书 www.zizidiary.com

表达式 assert(expr) 会计算 expr 的值。如果 expr 为非零值，那么不会产生任何效果；否则，assert 表达式会抛出一个类型为 AssertError 的异常。像 assert (expr, message) 这样的形式，则会将 message（必须可转换成字符串类型）作为包含在 AssertError 对象内的错误信息的一部分（如果 expr 为非零值，则 message 不会被计算）。在所有情况下，assert 自身的类型都为 void。

当你要构建一个完全高效的程序时，D 语言编译器提供了一个开关（对于参考实现 dmd，它是-release），它可用于忽略所有的 assert 表达式。因此，assert 应该被当作一种调试工具，而不是用于测试那些特殊条件的方法，这些条件即使失败也可能是情有可原的。基于同样的理由，将带有副作用的表达式放置在 assert 表达式内部，而如果程序的行为会依赖这些副作用，则这种做法是错误的。关于构建发行版本的更多细节，请参考第 11 章。

有些情况是按特殊的方式进行处理的：assert(false)、assert(0) 或者静态情形下已知 assert 操作为零值。断言功能总是被启用的（与构建标志无关），并且会调用可让进程突然停止运行的 HLT 机器代码指令。这个中断会提示操作系统生成一个内核转储（core dump）或者在有问题的那一行启动调试器。

先为逻辑 OR 表达式（请参考 2.3.15 节）作一个铺垫，总是会对表达式进行计算并对其结果进行断言的短语为 is(expr) || assert(false)。

第 10 章会深入讨论 assert 和其他能保证程序正确性的机制。

2.3.4.2　mixin 表达式

如果将表达式比喻成各式各样的螺丝刀，那么 mixin（混入）就像是一把带有可更换刀头的电动螺丝刀，像是可调节的离合器，或像是一个内置有无线摄像头和语音识别功能的脑部手术适配器。它实在太强大了！

简单地说，mixin 表达式允许你将一个字符串变成可执行代码。它的语法为 mixin(expr)，其中 expr 必须为一个编译时已知的字符串。这条限制规则让 mixin 没有了动态脚本的能力，例如，从终端读取并解释字符串。的确如此，D 语言不是解释性语言，编译器也不会让标准运行时支持这一项功能。也有一个好消息，即 D 语言在编译期间会运行一个功能稳定的解释器，你可以根据实际需要用它来构造字符串。

在编译期间，有能力操作字符串，并将它们转变成代码，这便有可能创建所谓的特定领域的嵌入式语言（其粉丝亲切地称其为 DSEL）。使用 D 语言来实现 DSEL 通常会完成这样一些操作：以字符串字面量的形式接收 DSEL 语句，在编译期间进行处理；以字符串的形式创建相应的 D 语言代码，并使用 mixin 将这个字符串转变成 D 语言。这个听起来有点不切实际，但 DSEL 是真实存在的。这里就有几个很不错的使用 DSEL 的实例：SQL 命令、正则表达式以及语法规范（如 yacc）。事实上，如果以前使用过 printf，那么你也就使用过 DSEL。printf 所使用的格式指示符实际上就是一种小语言，它专门用于描述文本数据的输出格式。

在不使用任何附加工具（语法解析器、连接器、代码生成器……）的情形下，D 语言允许你创

自在读书 www.zizidiary.com

建任何 DSEL。也就是说，标准库里的函数 bitfields（在模块 std.bitmanip 里）能接受位域定义，并且会生成最优的 D 语言代码来读取和写入它们，即使 D 语言本身并不支持位域也是可以的。

2.3.4.3　is 表达式

is 表达式能回答关于类型的质询（“Widget 这样的类型存在吗？”或者“Widget 是从 Gadget 继承来的吗？”），而且它是 D 语言强大的编译时自省功能的重要组成部分。所有 is 表达式的计算都是在编译期间完成的，并且会以一个布尔常量的形式返回质询答案。is 表达式有多种形式，如下所示。

（1）is(Type)和 is(Type Symbol)两种形式会检查类型是否存在。该类型可能是无效的，很多时候还是不存在的。例如：

```
bool
    a = is(int[]),      // 真，int[]是一个有效类型
    b = is(int[5]),     // 真，int[5]也是有效的
    c = is(int[-3]),    // 假，数组的大小无效
    d = is(Blah);       // 假，（如果 Blah 未定义）
```

在任何情况下，即使语义无效，类型 Type 在语法上也必须要正确。例如，is([]x[])在编译时会出错，因为它不是一个 false 常量。换句话说，可以只对语法上看起来像类型的内容进行查询。

如果存在符号 Symbol，则在结果为 true 的情况下，它会成为类型 Type 的一个别名。如果 Type 很长并且复杂，那么这种形式就特别有用了。还没介绍过的 static if 语句能够区分出真假情况。第 3 章会详细地讨论 static if，但基本的过程很简单——static if 会计算一个编译时表达式，然后编译由表达式为 true 时所控制的语句。

```
static if (is(Widget[100][100] ManyWidgets)) {
    ManyWidgets lotsOfWidgets;
    ...
}
```

（2）is(Type1 == Type2)和 is(Type1 Symbol == Type2)这两种形式在 Type1 与 Type2 相同时会得到 true 值。（它们可能因为使用了别名而拥有不同的名字。）

```
alias uint UInt;
assert(is(uint == UInt));
```

如果存在符号 Symbol，则在结果为 true 时，它会变成 Type1 的别名。

（3）is(Type1 : Type2)和 is(Type1 Symbol : Type2)这两种形式在 Type1 等同于 Type2 或者可以隐式转换到 Type2 时，会得到 true 值。例如：

```
bool
    a = is(int[5] : int[]),     // 真，int[5]可转换成 int[]
    b = is(int[5] == int[]),    // 假，它们是相同的类型
    c = is(uint : long),        // 真
    d = is(ulong : long);       // 真
```

同样地，如果存在符号 Symbol，则在结果为 true 时，它会变成 Type1 的别名。

（4）is(Type == Kind)和 is(Type Symbol == Kind)这两种形式会检查 Type 的类别。类别为下列关键字之一：struct、union、class、interface、enum、function、

自在读书 www.zizidiary.com

delegate 和 super。如果 Type 属于其中的某一个种类，则表达式结果为 true。如果存在 Symbol，则它会根据具体的类别进行定义，具体如表 2-4 所示。

表 2-4　is(Type Symbol == Kind)形式中 Symbol 的各类绑定

类别	Symbol 别名
struct	类型
union	类型
class	类型
interface	类型
super	基类（参见第 6 章）
enum	枚举的基类型（参见第 7 章）
function	函数类型
delegate	委托的函数类型
return	由函数、委托和函数指针返回的类型

2.3.4.4　括号表达式

括号可以改变惯用的优先级顺序：对于所有表达式‹expr›来说，(‹expr›)都是一个基本表达式。

2.3.5　后缀表达式

2.3.5.1　成员访问

成员访问操作符“a.b”会访问到对象或类型 a 里的名字为 b 的成员。如果 a 是一个复杂的值或类型，那么它也可以用括号括起来。b 也可以是 new 表达式（参见第 6 章）。

2.3.5.2　增值和减值

所有数字和指针类型都支持后增值操作符（lval++）和后减值操作符（lval--），其语义与 C 和 C++语言里的同名操作符类似：对 lval 进行增值和减值的计算，不过在修改之前，会先产生一个它的副本。lval 必须是一个左值。（与前增值和前减值操作符相关的内容，请参考 2.3.6.3 节。）

2.3.5.3　函数调用

熟悉的函数调用操作符 fun()会调用函数 fun。而 fun(<逗号分隔的列表>)会向 fun 传递一个参数列表。在 fun 被调用之前，所有参数都是按自左向右的顺序进行计算的。函数的类型必须与参数列表里值的类型和数目一致。如果函数定义时带上了@property 属性，那么通过单独指定函数的名称就可以不带任何参数调用该函数。通常 fun 就是所定义函数的名称，但它也是一个函数字面量（参见 2.2.7 节）或者一个能产生函数或委托指针的表达式。第 5 章会对函数进行详细描述。

2.3.5.4　索引

表达式 arr[i]会访问数组或关联数组 arr 的第 i 个（从 0 开始）元素。如果该数组不是

自在读书 www.zizidiary.com

关联的，那么 i 必须为整数类型。否则，i 必须能够转换成 arr 的键类型。如果索引表达式处于赋值操作符的左边（如 arr[i] = e），并且 arr 是一个关联数组，那么如果有一个元素不存在，此表达式会在数组里插入它。对于其他情况，如果 i 没有引用到 arr 里已有的元素，那么此表达式会抛出一个 RangeErrer 对象。如果 arr 是指针类型，而 i 为整数类型，则表达式 arr[i]也同样能工作。指针引用不会有边界检查。有些构建模式（发行不安全的构建，参见 4.1.2 节）可能还会针对非关联数组一起禁用边界检查。

2.3.5.5　数组分割

如果 arr 是一个线性（非关联）数组，那么表达式 arr[i .. j]会返回一个子数组，它会引用到 arr 内部从索引值 i 到 j（不包括）这段范围所确定的“窗口”。边界 i 和 j 必须能转换成整数类型。表达式 arr[]会获得 arr 的一个分割，它与 arr 自身一样大。实际的数据不会被复制，因此通过分隔操作符返回的数组，也会修改 arr 的内容。例如：

```
int[] a = new int[5];                    // 创建一个拥有 5 个整数元素的数组
int[] b = a[3 .. 5];                     // b 引用了 a 的最后面两个元素
b[0] = 1;
b[1] = 3;
assert(a == [ 0, 0, 0, 1, 3 ]);          // a 被修改了
```

如果 i>j 或 j>a.length，那么该操作符会抛出一个 RangeError 对象。如果 i==j，那么它会返回一个空数组。如果 arr 是指针类型，那么表达式 arr[i .. j]也能工作，并且会返回一个数组，而这个数据映射出的是从 arr + i 到 arr + j（不包括在内）这部分内存区域。如果 i>j，那么此索引操作会抛出一个 RangeError 对象；此外还有，对指针的范围分割不会进行边界检查。同样地，有些构建模式（发行不安全的构建，参见 4.1.2 节）可能会针对分割操作禁用边界检查。

2.3.5.6　创建嵌套类

形如 a.new T 的表达式（a 是一个类类型）会创建一个类型为 T 的对象，而 T 的定义被嵌套在 a 的定义里。如果觉得有些混乱，那是因为类里嵌套有类，此外连 new 表达式都也未被定义。关于 new 表达式的定义在 2.3.6.1 节介绍，但关于类和嵌套类的定义则需要等到第 6 章（参见 6.11 节）。本段只是占个位，出于完整性目的而被插入到这里。

2.3.6　一元表达式

2.3.6.1　new 表达式

new 表达式具有如下形式：

```
new (〈地址〉)可选 〈类型〉
new (〈地址〉)可选 〈类型〉(<参数列表>可选)
new (〈地址〉)可选 〈类型〉[<参数列表>]
new (〈地址〉)可选 〈匿名类〉
```

自在读书 www.zizidiary.com

先暂时忽略掉可选项“(‹地址›)”。前两种形式为“new T”和“new T(‹参数列表›可选)”，它们会动态分配一个类型为 T 的对象。第二种形式可以选择传递某些参数给 T 的构造函数。(“new T”和“new T()”是完全等同的，都会创建一个默认初始化的对象。)我们还未见到带有构造函数的类型，对它的讨论被推迟到了第 6 章的类（参见 6.6 节）以及第 7 章的其他自定义类型（参见 7.1.3 节）。匿名类分配（最后一种形式）也被推迟到了第 6 章（参见 6.11.3 节）。

这里，我们先把注意力集中到大家已熟悉的数组分配操作上。表达式 new T[n]会分配一块连续的内存（其空间大到足以一个挨一个地容纳下 n 个对象），并使用 T.init 来填满这些槽位，然后以 T[]值的形式返回一个针对它们的句柄。例如：

```
auto arr = new int[4];
assert(arr.length == 4);
assert(arr == [ 0, 0, 0, 0 ]); // 默认初始化了
```

使用稍微有些差异的语法也可以获得同样的结果：

```
auto arr = new int[](4);
```

这次，表达式被解释成 new T(4)，其中 T 为 int[]。而最终结果为一个通过 arr（其类型为 int[]）进行操作的包含有 4 个元素的数组。

第二种形式其实比第一种包含了更多的内容。如果要分配一个由数组构成的数组，则可以在括号里提供多个参数。它们会被当做维度的初始值，按行排序。例如，这里有一个实例，讲的是如何分配一个由 4 个数组构成的数组，而其中每一个数组拥有 8 个元素：

```
auto matrix = new int[][](4, 8);
assert(matrix.length == 4);
assert(matrix[0].length == 8);
```

上面代码片段里的第一行隐藏了很多细节：

```
auto matrix = new int[][](4);
foreach (ref row; matrix) {
    row = new int[](8);
}
```

所有分配的内存都是在垃圾回收堆（garbage-collected heap）上进行的。那些不再使用的或者应用程序无法访问的内存都会被自动回收。编译器参考实现 dmd 的运行时支持，在模块 core.gc 里提供了几个操控垃圾回收内存的基本工具，其中包括调整已分配内存块的大小以及手动释放内存等。手动管理内存存在一定的风险，除非真的有必要，否则应尽量不要使用。

在关键字 new 之后传入的可选地址 addr 会引入一个名叫“定点新建”（placement new）的结构。在此种情形下，new(addr)T 的语义有所不同：它不会为新的对象分配内存空间，而只是在指定的地址 addr 适时地创建一个对象。这是一个很底层的特性，并不会出现在普通代码里。例如，如果想要通过 malloc 在 C 语言的堆里分配内存，则可以使用它，然后使用它来存储 D 语言的值。

2.3.6.2　取地址和解引用

由于指针是后面要讨论的主题，因此，我们这里只是顺便提一下“取地址”和“解引用”（dereference）这一对操作符。对于类型为 T 的 lval，表达式&lval 会获得 lval（正如这个

自在读书 www.zizidiary.com

名字所暗示的，lval 必须是一个左值）的地址，并返回一个类型为 T*的指针。

操作符*p 可以对指针进行解引用，解出方式为：抵消掉&的作用，使得*&lval 等同于 lval。关于指针，第 7 章会有更详细的讨论，因为到那个时候，你应该可以在不依赖指针（它虽然卓越超群，但却属于很底层、很危险的特性）的前提下编写出很多优秀的 D 语言代码。

2.3.6.3　前增值与前减值

表达式++lval 和--lval 会分别对 lval 执行增值和减值的操作，并将 lval 刚才更改过的值作为结果提供。lval 必须为数字或指针类型。

2.3.6.4　按位求反

表达式~a 会对 a 里的每一位进行切换（反转）操作，而其类型与 a 相同。a 必须为整数类型。

2.3.6.5　一元加与一元减

表达式+val 并不会有什么明显的动作——它的存在仅仅是因为完备性的需要。表达式 -vat 会计算 0 - val 的值，并且只适应于数字类型。

关于一元减有一个令人惊讶的行为：当应用到一个无符号值上时，它仍然会产生一个无符号值（参照 2.3.3 节中进过的规则）。例如，-55u 为 4_294_967_241，即 uint.max - 55 + 1。

在现实中，无符号值也不是真正的自然数。在 D 语言和其他许多语言中，两者的补运算不可避免地具有自己的简单溢出规则，也是不可抹灭的事实。对于任何整数值 val，有一种关于 -val 的思考方法将它当成简短形式~val + 1。换言之，即对 val 里的每一个二进制位求反，然后将结果加 1。这样不会造成与 val 的符号有关的问题。

2.3.6.6　取反

表达式!val 的类型为 bool。如果 val 为非零值（参见 2.3.4.1 节关于非零值的定义），则会得到 false；否则，为 true。

2.3.6.7　cast 表达式

cast 操作符就像一位法力强大且存有好心的灯神，急于要去拯救光明。与神话里的灯神真的很像，cast（强制类型转换）很“淘气”、很不听话，而且擅长运用言词拙劣的“愿望”——要以很机械的方式才能满足它们，而且经常会造成灾难性的后果。

即是说，当静态类型系统不够灵活，以致无法满足你的应用时，强制类型转换会变得很有用。强制类型转换的语法为 cast(Type) a。有多种强制转换，现按安全性等级降序罗列如下。

- 引用强制转换允许在对类和接口对象的引用当中进行转换。这些转换总会动态地进行检查。
- 数字强制转换会将数据从任何数字类型强制为其他任何数字类型。
- 数组强制转换允许不同数组类型之间的转换，只要源数组的总大小能被目标数组元素的大小整除即可。
- 指针强制转换会接受一个指向某种类型的指针，并将它转换成指向另一种类型的指针。

自在读书 www.zizidiary.com

- 指针数字强制转换会将一个指针转换成一个足以容纳下它的整数类型，而且允许反向操作。

请特别注意所有这些不会进行检查的强制类型转换，尤其是最后三点，它们可能会影响类型系统的完整性。

2.3.7　幂表达式

幂表达式的形式为 base ^^ exp，它会计算出 base 的 exp 次方值。base 和 exp 都必须是数字类型。它所提供的功能与 C 语言和 D 语言标准库（请参考关于模块 std.math 的文档）里的库函数 pow(base , exp) 是一样的。不过，有些数值应用程序会从这种简化语法的版本中获益。

0 的 0 次方为 1，而 0 的任何非 0 次方为 0。

2.3.8　乘法表达式

乘法表达式有：相乘（a * b）、相除（a / b）以及求模（a % b）。它们都只针对数字类型。

在整数运算 a / b 或者 a % b 中，如果 b 为 0，那么会有一个硬件中断抛出。如果除法产生一个小数，那么它总是会朝 0 的方向截断（例如，7 / 3 为 2，而−7 / 3 为−2）。定义了表达式 a % b 之后，就会有 a == (a / b) * b + a % b，因此 7 % 3 为 1，而−7 / 3 为−1。

针对浮点数，D 语言还定义了求模运算，其定义包含了很多内容。在 a % b 里，当 a 和 b 至少有一个是浮点值时，其结果为满足下列条件的最大（绝对值）浮点数 r。

- a 和 r 的符号相同。
- r 比 b 的绝对值更小，即 abs(r) < abs(b)。
- 存在有一个整数 q，使得 r == a - q * b 成立。

如果无法找到这样的数，那么 a % b 会产生一个特殊值 NaN（Not A Number，不是一个数）。

2.3.9　加法表达式

加法表达式有：相加 a + b、相减 a - b 以及连接 a ~ b。

加法和减法运算只针对数字类型。结果类型的确定方法在 2.3.3 节有描述。

连接运算要求 a 和 b 至少有一个为数组类型，数组包含元素的类型为 T。其他值也必须为一个 T 类型的数组或者一个可隐式转换为 T 类型的值。其结果是一个由 a 和 b 并列组成的新数组。

2.3.10　移位表达式

在 D 语言里有 3 种移位运算，它们都接受两个整数值：a << b、a >> b 和 a>>>b。在

自在读书 www.zizidiary.com

所有情形里，b 都必须为一个无符号类型。如果都是有符号值，则必须将 b 强制类型转换成一个无符号类型（尽可能保证 b >= 0，移位一个负的量会得到不确定的值）。a << b 会将 a 向左（即往 a 的最高有效位方法）移动 b 个二进制位；而 a>>b 则会将 a 向右移动 b 个二进制位。如果 a 为负值，则移位操作会保留其符号。

a>>>b 是一个无符号的移位操作，a 的符号没有意义。即是说，0 可能会被移到 a 的最高符号位，也确实是这样。下面举例说明针对有符号数的移位操作有哪些令人惊讶的效果。

```
int a = -1;                    // 即 0xFFFF_FFFF
int b = a << 1;
assert(b == -2);               // 0xFFFF_FFFE
int c = a >> 1;
assert(c == -1);               // 0xFFFF_FFFF
int d = a >>> 1;
assert(d == +2147483647);      // 0x7FFF_FFFF
```

如果 b 是一个静态已知的值，则在编译期间，不允许在移动一定数量的位后，其值超过 a 的类型所允许的范围；如果 b 是一个运行时值，则它会在 a 里放置一个与编译器实现相关的值。

```
int  a = 50;
uint b = 35;
a << 33;                       // 编译时错误
auto c = a << b;               // 实现所定义的结果
auto d = a >> b;               // 实现所定义的结果
```

在所有情况下，结果类型的确定都会根据 2.3.3 节所描述的规则。

以前，移位操作的流行用法是代价很低的整数乘以 2（a<<1）或除以 2（a>>1），或者更一般地，乘以和除以 2 的各种次方。这种技术已经过时，现在只需写成 a * k 或者 a / k 即可。只要 k 是一个编译时已知的量，那么编译器会生成带移位操作的优化代码，同时不用担心符号是否正确的问题。时代变了！

2.3.11　in 表达式

如果 key 的类型为 K，map 是一个类型为 V[K]的关联数组，那么表达式 key in map 会产生一个类型为 V*（V 类型指针）的值。如果该关联数组包含有<key, val>这样的一对值，那么该指针会指向 val；否则，该指针为 null。

对于反向负值测试，当然可以写成!(key in map)，也可以使用更简洁的形式 key !in map——它与 key in map 拥有相同的优先级。

为什么要使用指针，而不让 a in b 产生一个布尔值？其目的在于提高效率。人们时常会想要查询关联数组里的索引值，如果映射后的元素已经存在，则可以直接使用它。可能出现的情况如下：

```
double[string] table;
...
if ("hello" in table) {
   ++table["hello"];
} else {
   table["hello"] = 1;
}
```

自在读书 www.zizidiary.com

上面这段代码的问题在于，它在成功的那个分支上执行了两次查询。如果使用返回指针的方法，那么这段代码可以写成更有效率的版本，如：

```
double[string] table;
...
auto p = "hello" in table;
if (p) {
   ++*p;
} else {
   table["hello"] = 1;
}
```

2.3.12　比较操作符

2.3.12.1　相等比较

a == b，其类型为 bool，具有如下语义：首先，如果两个操作数的类型不相同，那么执行隐式转换，使它们具有相同的类型；然后，两个操作数按照下面的要求进行相等性比较。

- 对于整数类型和指针，相等性的定义是所有值都在位模式下按位进行精确比较。
- 对于浮点数类型，-0 等于+0，而 NaN 不等于 NaN[①]。对于其他所有情况，相等性的定义都是要求位模式相等。
- 对于类对象，相等性的定义需要借助于 opEquals 操作符（参见 6.14 节）。
- 对于数组，相等性表示的是元素对元素相等。
- 对于结构对象，相等性默认定义为域对域的相等。对于自定义类型可能会改写此行为（参见第 12 章）。

a != b 形式可用于测试不等性。

表达式 a is b 比较的是别名。如果 a 和 b 都实际引用了同一个对象，则返回 true。

- 如果 a 和 b 都是数组或类的引用，那么只有当 a 和 b 实际都是同一个对象的两个名称时，结果才为 true。
- 否则，a is b 等同于 a == b。

目前我们对类的了解不多，不过可以使用带数组的示例来帮助理解：

```
import std.stdio;

int main(){
   auto a = "some thing";
   auto b = a; // a 和 b 都引用了同一个数组
   a is b && writeln("Yep, they're the same thing really");
   auto c = "some (other) thing";
   a is c || writeln("Indeed... not the same");
}
```

① IEEE 754 浮点数对于零有两种不同的位模式，分别定义为-0 和+0。它们引起了一些小烦恼，如这里指出的特殊情况，但也加快了很多计算。在 D 语言里，应尽量少用-0.0 字面量，但它可能会以计算结果的形式偷偷地产生，此类计算主要带有负值并逐渐地逼近零。

自在读书 www.zizidiary.com

因为 a 和 b 实际都绑定到同一个数组，而 c 绑定的是不同的数组，所以上面的代码会输入两条信息。一般情况下，两个数组的内容可能会相等（即 a == c 为 true），但它们却指向了不同的内存区域，因此测试 a is c 会失败。当然，如果 a is c 为 true，很显然 a == c 也会为 true，除非你的 RAM 芯片是以非常非常便宜的价格购买的。

不等性操作符 a !is b 可以改写成 !(a is b)。

2.3.12.2　顺序比较

D 语言定义的表达式有 a < b、a <= b、a > b 和 a >= b，类型为 bool，一般的含义分别为小于、小于等于、大于和大于等于。当进行数字比较时，它们当中某一个的类型必须要能隐式转换成另一个的类型。对于浮点操作数，-0 和 0 被认为是相等的，因此-0 < 0 得到 false，而 0 <= -0 会得到 true。无论什么时候，只要它们当中有一个操作数为 NaN（看起来有些荒谬），那么全部的顺序比较操作符都会返回 false。

一如既往，当很顺利地处理合法的浮点数时，NaN 总是会把事情弄得一团糟。所有比较操作符只要接收到 NaN 值，便会抛出一个浮点数异常。在常见的编程语言术语里，它不是一个真正的异常，而是一个可以进行显式地检查的硬件级的状态。D 语言通过模块 std.c.fenv 提供了一个针对浮点硬件的接口。

2.3.12.3　非结合性

所有 D 语言的比较操作符都有一个重要的特征，即它们都是不可结合的。任何链式的比较操作（如 a <= b < c），都是非法的。

有一种简单的定义比较操作符的方法，即让它们的结果为布尔类型。布尔值自身能够进行比较，这个可能会造成令人遗憾的状况：a <= b < c 并不具有少数数学家所期望的含义，我们大家都在努力摆脱这种情况。如果真的采用此方法，那么这个表达式就可以解析成 (a <= b) < c，即“将由 a<=b 得到的布尔结果与 c 进行比较”。例如，3 <= 4 < 2 的结果就会是 true！这绝不是你想要的语义。

还有一种可能的解决方案，它可以支持 a <= b < c 并赋予它更直观的数学含义，即 a <= b && b < c，并让 b 只计算一次。像 Python 和 Perl 6 这样的语言已支持这种语义，它们支持像 a < b == c > d < e 这样的随意比较链。不过，D 语言没有去完全继承。如果让 D 语言支持 C 语言表达式，但却使用稍微有些不同的语义（可以说是朝着正确的方向），那么引起的混乱会远多于带来的方便，因此 D 语言断然地拒绝了这个构想。

2.3.13　按位操作符

表达式 a | b、a ^ b 和 a & b 可分别用于进行或（OR）、异或（XOR）和与（AND）的按位运算。即使结果完全可能通过一边判定出来，两边也都会进行计算（不会缩短操作）。

自在读书 www.zizidiary.com

a 和 b 都必须是整数类型。结果的类型需要根据 2.3.3 节来确定。

2.3.14　逻辑与

按照上面的内容，表达式 a && b 的语义依赖于 b 的类型一点都不会让人觉得惊讶。

- 如果 b 的类型不是 void，那么表达式的类型为 bool。如果 a 为非零值，则表达式会计算 b 的值，并且仅当 b 为非零值时才会得到 true。否则，表达式计算结果为 false。
- 如果 b 的类型为 void，那么此表达式的类型也为 void。如果 a 为非零值，则需要计算 b 的值。否则，不计算 b。

在 void 型表达式的右边使用&&能够缩短 if 语句：

```
string line;
...
line == "#\n" && writeln("Got a # successfully");
```

2.3.15　逻辑或

表达式 a || b 的语义依赖于 b 的类型。

- 如果 b 的类型不是 void，那么表达式的类型为 bool。如果 a 为非零值，则表达式计算结果为 true。否则，表达式计算出的值为 b 的值，并且仅当 b 为非零值时才会得到 false。
- 如果 b 的类型为 void，那么此表达式的类型也为 void。如果 a 为非零值，则不需要计算 b。否则，计算 b 的值。

第二点对意外情况的处理比较有用：

```
string line;
 ...
line.length > 0 || line = "\n";
```

2.3.16　条件操作符

条件操作符是一个 if-then-else 表达式，其语法为 a ? b : c。你可能对它已很熟悉了。如果 a 为非零值，则条件表达式会计算 b 并产生结果；否则条件表达式会计算 c 并产生结果。编译器付出了很大的努力才找到关于 b 和 c 的最紧凑的公共类型，这便是条件表达式的类型。此类型（假定为 T）是通过一个很简单的算法（如下例所示）计算得到的：

（1）如果 b 和 c 的类型相同，则 T 也为该类型；

（2）如果 a 和 b 都是整数，则首先将所有小于 32 位的类型提升至 int 类型，接着让 T 选择更大的那一个类型（如果大小受限，则优先选择无符号类型）；

（3）如果有一个为整数，而另一个为浮点类型，则 T 为浮点类型；

自在读书 www.zizidiary.com

（4）如果两者都为浮点类型，则 T 为两者之间更大的那一个；

（5）如果这些类型有一个公共的超类型（如基类），那么 T 便是该超类型（在第 6 章会再回到这个话题）；

（6）接着尝试将 b 隐式转换至 c 的类型以及将 c 隐式转换到 b 的类型；如果恰好有一个成功了，则 T 即为成功转换后的目标类型；

（7）否则，表达式出错。

此外还有，如果 b 和 c 都拥有相同的类型，而且都是左值，那么其结果也是一个左值，并可以写成这样：

```
int x = 5, y = 5;
bool which = true;
(which ? x : y) += 5;
assert(x == 10);
```

有很多泛型编程惯用法都使用了条件操作符来访问两个值的共同类型。

2.3.17　赋值表达式

赋值表达式的形式有 a = b 或者 a ω= b，这里的 ω 代表的是这样一些操作符号：^^、*、/、+、<<、>>、>>>、|、^、&以及“编程语言书籍里强制使用的希腊字母”。前面几节已单独介绍过了这些操作符。

a ω= b 的语义等同于 a = a ω b，其显著的不同点在于 a 只会计算一次。想象一下，a 和 b 可能是很复杂的表达式，如 array[i * 5 + j] *= sqrt(x)。

ω 的优先级可以不管，而 ω=的优先级与=是一样的，刚好在上面讨论过的条件操作符之下，而比下面将要讨论的逗号操作符的优先级稍高一点。同样，ω 的结合性也可以忽略，所有的 ω= 操作符（还有=）都可以自右向左结合在一起。如 a /= b = c -= d 与 a /= (b = (c -= d)) 是完全一样的。

2.3.18　逗号表达式

逗号分隔的表达式会依次进行计算。整个表达式的结果为最右边那个表达式的结果。例如：

```
int a = 5;
int b = 10;
int c = (a = b, b = 7, 8);
```

在执行上面的代码片段之后，a、b 和 c 的值分别为 10、7 和 8。

2.4　小结和快速参考

这里总结的是 D 语言丰富的表达式构建方式。表 2-5 归纳了 D 语言的全部操作符，需要的时候都可以随时到这里来查阅。

自在读书 www.zizidiary.com

表 2-5　按优先级递减排列的表达式

表达式	描述
‹符号›	符号（参见 2.1 节）
.‹符号›	访问模块作用域里的符号（忽略其他所有作用域）（参见 2.1 节）
this	在一个方法内部的当前对象（参见 2.1 节）
super	通过基子对象来指导符号查找和动态方法查找（参见 2.1 节）
$	当前数组的大小（在索引或分割表达式内有效）（参见 2.1 节）
null	空引用（用于数组或指针）（参见 2.1 节）
typeid(T)	获得与 T 相关联的 TypeInfo 对象（参见 2.1 节）
true	布尔真值（参见 2.2.1 节）
false	布尔假值（参见 2.2.1 节）
‹数字›	数字字面量（参见 2.2.2 节）
‹字符›	字符字面量（参见 2.2.4 节）
‹字符串›	字符串字面量（参见 2.2.5 节）
‹数组›	数组字面量（参见 2.2.6 节）
‹函数›	函数字面量（参见 2.2.7 节）
assert(a)	在调试模式下，如果 a 为一个非零值，则中断程序；在发行模式下，什么也不做（参见 2.3.4.1 节）
assert(a, b)	同上，另外会将 b 作为出错信息的一部分（参见 2.3.4.1 节）
mixin(a)	mixin 表达式（参见 2.3.4.2 节）
‹IsExpr›	is 表达式（参见 2.3.4.3 节）
(a)	括号表达式（参见 2.3.4.4 节）
a.b	成员访问（参见 2.3.5.1 节）
a++	增值（参见 2.3.5.2 节）
a--	减值（参见 2.3.5.2 节）
a(‹*csl* 可选›)	函数调用操作符（‹*csl* 可选›为可选的以逗号分隔的参数列表）（参见 2.3.5.3 节）
a[‹*csl*›]	索引操作符（‹*csl*›为以逗号分隔的参数列表）（参见 2.3.5.4 节）
a[]	分割整个集合（参见 2.3.5.5 节）
a[b .. c]	分割（参见 2.3.5.5 节）
a.‹new 表达式›	创建嵌套的实例（参见 2.3.5.6 节）
&a	取地址（参见 2.3.6.2 节）
++a	增值（参见 2.3.6.3 节）
--a	减值（参见 2.3.6.3 节）
*a	解引用（参见 2.3.6.2 节）
-a	一元减（参见 2.3.6.5 节）
+a	一元加（参见 2.3.6.5 节）
!a	取反（参见 2.3.6.6 节）
~a	按位求补（参见 2.3.6.4 节）
(T).a	静态成员访问
cast(T)a	强制类型转换表达式
‹new 表达式›	对象创建（参见 2.3.6.1 节）
a ^^ b	求幂（参见 2.3.7 节）
a * b	乘（参见 2.3.8 节）

自在读书 www.zizidiary.com

续表

表达式	描述
a / b	除（参见 2.3.8 节）
a % b	求模（参见 2.3.8 节）
a + b	加（参见 2.3.9 节）
a - b	减（参见 2.3.9 节）
a ~ b	连接（参见 2.3.9 节）
a << b	左移（参见 2.3.10 节）
a >> b	右移（参见 2.3.10 节）
a >>> b	无符号右移（不管 a 的类型和值是什么，结果的最高符号位都为零）（参见 2.3.10 节）
a in b	关联数组的隶属关系测试（参见 2.3.11 节）
a == b	相等性测试。这组里的所有操作符都不可结合，例如：a == b == c 便是非法的（参见 2.3.12.1 节）
a != b	不等性测试（参见 2.3.12.1 节）
a is b	一致性测试（仅当 a 和 b 都引用到同一对象时才为真）（参见 2.3.12.1 节）
a !is b	等同于!(a is b)
a < b	小于（参见 2.3.12.2 节）
a <= b	小于等于（参见 2.3.12.2 节）
a > b	大于（参见 2.3.12.2 节）
a >= b	大于等于（参见 2.3.12.2 节）
a \| b	按位或（参见 2.3.13 节）
a ^ b	按位异或（参见 2.3.13 节）
a & b	按位与（参见 2.3.13 节）
a && b	逻辑与（b 可以为 void 类型）（参见 2.3.14 节）
a \|\| b	逻辑或（b 可以为 void 类型）（参见 2.3.15 节）
a ? b : c	条件操作符。如果 a 为非零值，则为 b，否则为 c（参见 2.3.16 节）
a = b	赋值。在本组里的所有赋值操作符都可以自右向左绑定一定，例如，a *= b += c 等同于 a *= (b += c)（参见 2.3.17 节）
a += b	就地加：在此组里的所有“计算并赋值”操作符 a ω= b 按这样的顺序进行计算：（1）计算 a（它必须是一个左值）；（2）计算 b；（2）计算 a_1 = a_1 ω b，这里的 a_1 是计算 a 之后得到的左值
a -= b	就地减
a *= b	就地乘
a /= b	就地除
a %= b	就地除
a &= b	就地按位与
a \|= b	就地按位或
a ^= b	就地按位异或
a ~= b	就地连接（将 b 添加到 a）
a <<= b	就地左移
a >>= b	就地右移
a >>>= b	就地无符号右移
a , b	此表达式会自左向右进行计算，其结果为最右边那个表达式计算出的值（参见 2.3.18 节）

自在读书 www.zizidiary.com

第3章

语句

本章内容为一些强制性定义，其中涉及了 D 语言定义的所有语句。D 语言构建于 C 语言系列外观之上——都有 if、while、for 等语句。不过 D 语言还包含了一些新的、有意思的语句，并且对原有的一些语句进行了微调。在对每一条语句都进行不可避免的、冗长的详细描述时，为防止大家觉得乏味，这里先列出一些“变化点”——这可是 D 语言很不错的创新点。

如果需要按条件编译代码，那么 static if（参见 3.4 节）一定会是你感兴趣的内容。除了简单的以标志块实现定制的功能外，它还有很多用途。在任何情况下，当使用泛型代码时，static if 都能带来巨大的好处。switch 语句（参见 3.5 节）与 C 语言里同一条语句没什么两样，只是在 D 语言里它还能用于字符串，并且支持范围匹配。如果想正确地处理小规模的闭集值，那么 final switch（参见 3.6 节）语句能带来更多好处。它可以与枚举类型一起工作，并且会强制你对每一个可能值进行处理。foreach 语句（参见 3.7.4 节至 3.7.5 节）对于连续迭代操作非常方便，但经典的 for 更为通用，只是它有些稍显冗长。mixin 语句（请参考 3.12 节）扩展了预定义样板代码的功能。scope 语句（参见 3.13 节）通过替换复杂的 try/catch/finally 语句（在没有 scope 语句时，你可能不得不编写它们），大大简化了带有正确的错误恢复功能的正确事务代码的编写。

3.1 表达式语句

如前所述（请参见 1.2 节），如果在表达式的后面添加一个分号，它就变成了语句：

```
a = b + c;
transmogrify(a + b);
```

不过，不是所有的表达式都能成为语句。如果产生的语句没有意义，则编译器会给出一个错误提示。例如：

```
1 + 1 == 2; // 不变的真理
```

自在读书
www.zizidiary.com

3.2　复合语句

复合语句指的是封闭在大括号里的一系列语句（也可能为空）。这些语句会按次序执行。大括号会引入一个词法作用域：在复合语句内部定义的符号在其外部是不可见的。

在作用域内部定义的符号会掩盖掉处于所有作用域之外的同形异义的符号：

```
uint widgetCount;
...
void main() {
    writeln(widgetCount);   // 写入到全局符号
    auto widgetCount = getWidgetCount();
    writeln(widgetCount);   // 写入到本地符号
}
```

第一个 writeln 调用会输出全局符号 widgetCount，而第二个 writeln 调用会访问本地定义的 widgetCount。有时也需要访问那些被掩盖的全局符号，则可以按照 2.1 节所说的，在其前面添加一个点，如 writeln(.widgetCount)。不过，如果新定义的符号会掩盖封闭在复合语句内部的符号，则是不合法的：

```
void main() {
    auto widgetCount = getWidgetCount();
    // 现在开始一个嵌套块
    {
        auto widgetCount = getWidgetCount(); // 出错!
    }
}
```

只要未发生掩盖，在不同复合语句里再次使用相同的符号，则是合法的：

```
void main() {
    {
        auto i = 0;
        ...
    }
    {
        auto i = "eye"; // 可以
        ...
    }
    double i = 3.14;    // 也可以
}
```

这种设置的原理很简单。允许掩盖全局符号对于编写优秀的模块化的代码（它能将各个单独编译的部分汇集到一起）是很有必要的。因添加一个全局变量而导致大量无辜的不相干代码无法编译，这是大家都不想要的。另一方面，让封闭作用域的掩盖成为一个模块化装置（在 D 语言里，还从未有过一个复合语句跨越多个模块的情况）也没有用处，大部分情况表明要么是由于疏忽导致错漏，要么就是“癌变”函数失去控制。

3.3　if 语句

各种类型的实例都已经用到了 D 语言的 if 语句，它与你想象的样子差不多：

```
if (‹表达式›) ‹语句 1›
```

自在读书 www.zizidiary.com

或者

```
if (<表达式>) <语句1> else <语句2>
```

关于 if 所控制的语句没什么好说的。与其他语言不同，D 语言没有“空语句”的概念。特别地，只带有一个分号时，它也不是一条语句，这种情况会标记为一个错误。这种设计可以让程序员避免发生类似下面这样的错误：

```
if (a == b);
    writeln("a and b are equal");
```

在表达式很短或者还能掌控它的时候，这类代码的问题还比较明显；但是，当表达式很长时，整个结构被成片的代码淹没时，以及在凌晨两点的时候，情况没就那么简单了。如果想要使用 if 来控制一个空语句，则可以使用最近似于空语句的方法——内部没有任何语句的复合语句：

```
if (a == b) {}
```

当你重构代码并且偶尔会对代码的各个部分进行注释和取消注释时，这种方法会非常有用。

else 子句总是与最靠近它的那个 if 绑定在一起，因此下面代码的缩进是正确的：

```
if (a == b)
   if (b == c)
      writeln("All are equal");
   else
      writeln("a is different from b. Or is that so?");
```

第二个 writeln 会在 a == b 且 b != c 时执行，这是因为与 else 绑定在一起的是最里面那个（第二个）if。如果想让 else 绑定到第一个 if，则可以使用一对大括号来“缓冲”第二个 if：

```
if (a == b) {
   if (b == c)
      writeln("All are equal");
}
   else
       writeln("a is different from b");
```

级联式的多重 if-else 语句可以借用经久不衰的 C 语言样式来完成：

```
auto opt = getOption();
if (opt == "help") {
   ...
} else if (opt == "quiet") {
   ...
} else if (opt == "verbose") {
   ...
} else {
   stderr.writefln("Unknown option '%s'", opt);
}
```

3.4　static if 语句

现在我们已经熟悉了一些简单的语句，接下来再看一些没那么普通的内容。

如果想要根据编译时的布尔条件来“注释掉”（或者不注释）某些语句，那么可以借助于 static if 语句[①]。例如：

① 打断一下，“这是关键字 static 的又一种用法”。

自在读书 www.zizidiary.com

```
enum size_t
   g_maxDataSize = 100_000_000,
   g_maxMemory = 1_000_000_000;
...
double transmogrify(double x) {
   static if (g_maxMemory / 4 > g_maxDataSize) {
      alias double Numeric;
   } else {
      alias float Numeric;
   }
   Numeric[] y;
   ... // 复杂的计算
   return y[0];
}
```

static if 语句是一个编译时选择器，有点像 C 语言里的#if 指令。当遇到一个 static if 时，编译器会计算控制表达式。如果表达式为非零值，那么编译相应的代码；否则，编译与 else 子句（如果有的话）对应的代码。在上面的示例里，static if 被用来实现在节省内存操作模式（使用更小的 float）与精确模式（使用更精确的 double）之间的切换。它在泛型代码里的应用会表现得更为强大和更具表现力。

static if 测试的表达式可以是任何能使用 if 进行测试的表达式，并且它还要能在编译期间进行计算。支持的表达式涉及了语言的大量子集，其中包括用于所有数字类型的算术运算、数组操作、关于类型的 is 表达式运算（参见 2.3.4.3 节），甚至还包括函数调用——这是 D 语言绝对耀眼的特性，也被称为编译时函数计算。在第 5 章会深入讨论编译时计算。

剥掉括号　在关于 transmogrify 的示例里，有一个很奇特的地方，即在括号“{”和“}”内部引入了一个 Numeric 类型。按理说，它应该只在这个作用域内才是可见的（即对于这个闭合函数是不可见），看来我们的整个计划就此泡汤了！这种行为也会导致事先承诺的 static if 语句毫无用武之地。基于这个原因，规定 static if 使用括号只是为了分组，而不是要引入作用域。就作用域和可见性而言，如果存在最外层的括号（当只有一个控制语句时，它们就不是必需的；在上面的示例里，它们只起到改善语句排列的作用），static if 会将它们剥掉。如果实在想要大括号，只须另外添加一对即可：

```
import std.stdio;

void main() {
   static if (real.sizeof > double.sizeof) {{
      auto maximorum = real.max;
      writefln("Really big numbers - up to %s!", maximorum);
   }}
   ... /* maximorum 在这里不可见 */ ...
}
```

不是语句的语句　本章的标题是什么？“语句”。本节的标题是什么？“static if 语句”。static if 不仅仅是语句，也是一个声明。如果这一点让人觉得有些意外，也是很正常的。static if 的非语句特性不仅仅体现在大括号的剥离行为上，还存在于这样一种实际情况里：static if 可以出现在声明能出现的任何地方，也可以出现在模块、结构和类这样的层次上，而这些地方是语句无法访问到的。例如，如下所示，采用将有关的代码从 transmogrify 中提取出来的简单方式，便可以定义全局 Numeric：

自在读书 www.zizidiary.com

```
enum size_t
   g_maxDataSize = 100_000_000,
   g_maxMemory = 1_000_000_000;
...

// 声明在模块作用域可见的 Numeric
static if (g_maxMemory / 4 > g_maxDataSize) {
   alias double Numeric;
} else {
   alias float Numeric;
}

double transmogrify(double x) {
   Numeric[] y;
   ... // 复杂的计算
   return y[0];
}
```

两个 if，一个 else　static if 没有与之对应的 static else。事实上，它只是重新利用了常规的那个 else。在逻辑上，else 只会绑定到与之最近的那个 if，不管它是静态的还是常规的：

```
if (a)
   static if (b) writeln("a and b are nonzero");
   else writeln("b is zero");
```

3.5　switch 语句

说明 switch 的最好方法就是来看一个简单的示例：

```
import std.stdio;

void classify(char c) {
   write("You passed ");
   switch (c) {
      case '#':
         writeln("a hash sign.");
         break;
      case '0': .. case '9':
         writeln("a digit.");
         break;
      case 'A': .. case 'Z': case 'a': .. case 'z':
         writeln("an ASCII character.");
         break;
      case '.', ',', ':', ';', '!', '?':
         writeln("a punctuation mark.");
         break;
      default:
         writeln("quite a character!");
         break;
   }
}
```

switch 语句的一般形式为：

```
switch (‹表达式›) ‹语句›
```

‹表达式›可以是数字、枚举和字符串。‹语句›可以包含如下几种定义标签。

（1）case ‹e›:

如果‹表达式› == ‹e›，则跳转到这里。想要在 e 里使用逗号操作符（参见 2.3.18 节），则需要将整个表达式使用圆括号括起来。

自在读书 www.zizidiary.com

(2) case ‹e_1›, ‹e_2›, ..., ‹e_n›:

每一个‹e_k›都是一个表达式。此结构等同于 **case ‹*item*$_1$›: case ‹*item*$_2$›:, ..., case ‹*item*$_n$›:**。

(3) case ‹e_1›: .. case ‹e_2›:

如果‹表达式› >= ‹e_1›并且‹表达式› <= ‹e_2›，则跳转到这里。

(4) default:

如果没有发生其他跳转则跳转到这里。

在所有这些测试中，‹表达式›只会计算一次。每一种情况标签里的表达式，可以是任何合法的表达式，且可以与‹表达式›进行相等比较；如果使用了“..”语法，则执行不等比较。通常情况下，case 表达式是编译时常量，不过 D 语言也支持变量，并且会保证以词法顺序进行计算，直到第一个匹配。在计算之后，会跳转到相应的 case 或 default 标签，并从那里开始执行过程。与 C 和 C++语言一样，switch 内部的执行在到达下一个情况标签时不会自动停止。如果想停止，则必须插入一个 break 语句。之所以决定保留这种行为，目的在于不想让从其他语言转过来的程序员感觉到不适应。

在编译期间计算的标签，不允许出现重叠情况。例如，下面的代码是非法的：

```
switch (s) {
   case 'a' .. case 'z': ... break;
      // 特殊情况'w'
   case 'w': ... // 出错！情况标签不能出现交错！
}
```

如果最终没发生跳转，则会抛出一个运行时异常。这是为了防止像“值的子集被无意中忽视掉”这类常见的编程错误。如果不存在这种风险，则可以在受控语句里插入一个 default:break;，这样对于设定情况就有了很好的文档说明。下一节会讲到在相同条件下的静态增强版本。

3.6 final switch 语句

经常会有这样的情况：将 switch 与枚举类型一起使用，即它要去处理所有可能的值。在维护期间，如果情况的数目发生了变化，则所有依赖的 switch 语句都无法立即保持同步，并且必须进行手工搜索和修改。

无疑，具有伸缩性的解决方案是使用虚函数调度来代替基于标签的开关操作。这样就没必要在一个地方处理所有的情况，而是可以将处理分布于不同的接口实现中。不过，现实有时是很残酷的。类和接口的定义前期需要付出很大的努力，而这正是基于 switch 的解决方案能够避免的。在这种情况下，final switch 便能派上用场。它可以静态强制 case 标签必须要覆盖枚举类型的所有可能值：

```
enum DeviceStatus { ready, busy, fail }
...
void process(DeviceStatus status) {
   final switch (status) {
      case DeviceStatus.ready:
         ...
      case DeviceStatus.busy:
```

自在读书 www.zizidiary.com

```
            ...
        case DeviceStatus.fail:
            ...
    }
}
```

假设在代码维护时添加了另一个可能的设备状态：

```
enum DeviceStatus { ready, busy, fail, initializing /* 增加的 */ }
```

在更改之后，试试重新编译 process，则会遇到类似下面那样的拒绝信息：

```
Error: final switch statement must handle all values
```

final switch 语句会要求 enum 的所有标签都被显式地处理。它不允许有基于范围的 case 标签形式 case ‹e_1›: .. case ‹e_2›:和 default 标签。

3.7 循环语句

3.7.1 while 语句

没错，你猜的绝对正确！就是这样的语句：

```
while (‹表达式›) ‹语句›
```

上面的语句在执行开始时会计算‹表达式›的值。如果它为非零值，则执行‹语句›，接着再次计算‹表达式›，恢复循环；否则，结束 while 语句，并执行后面的语句。

3.7.2 do-while 语句

如果需要至少执行一次循环，则可以借助于 do-while 语句：

```
do ‹语句› while (‹表达式›);
```

请注意，在这种结构后面，那个分号是必不可少的。还有，‹语句›也不能为空。do-while 语句与 while 语句一样，都会强制执行一次‹语句›。

3.7.3 for 语句

for 语句的语法为：

```
for (‹声明表达式› ‹表达式1›; ‹表达式2›) ‹语句›
```

‹声明表达式›、‹表达式 1›和‹表达式 2›都可以省略。如果省略了‹表达式 1›，则它会被认为是 true。‹声明表达式›结构可以引入一个值声明（如 auto i = 0;或 float w;），也可以是一个后面紧跟有“;”的表达式（如 i = 10;）。此语句的语义与其他语言里的同名结构语义相似：首先，它会计算‹声明表达式›，然后‹表达式 1›；只要它为真，则执行‹语句›，接着是执行‹表达式 2›，之后通过再次计算‹表达式 1›继续这个周期。

自在读书 www.zizidiary.com

3.7.4 foreach 语句

最方便、最安全和常常都是最快的迭代方式为 foreach 语句，它有好几种方式。foreach 最简单的形式为：

```
foreach (‹符号›; ‹表达式1› .. ‹表达式2›) ‹语句›
```

两个表达式都必须为数字或指针类型。简单来说，‹符号›会遍历‹表达式 1›（包括）到‹表达式 2›（不包括）这段区间。这种非正式的解释虽然很简洁，但却忽略了很多细节。例如，在整个过程中，‹表达式 2›是只执行一次还是会执行多次呢？或者如果‹表达式 1› >= ‹表达式 2›会发生什么呢？通过查看下面语义上等效的代码，这些细节很容易就显现出来。有一种技术叫做“降级”（lowering），它指的是借助更为简单的（子）语言里的概念来表达更高级的概念。本章会大量用到它。

```
{
    auto __n = ‹表达式2›;
    auto ‹符号› = true ? ‹表达式1› : ‹表达式2›;
    for (; ‹符号› < __n; ++‹符号›) ‹语句›
}
```

这里的__n 是编译器生成的一个符号，它保证不会与其他符号出现冲突（按照编译器作者的行话，它就是一个“新鲜”符号）。

外层的那对大括号起什么作用呢？它们保证‹符号›不会跑到 foreach 语句外面去，同时也保证整个结构是一条语句。

现在清楚了，‹表达式 1›和‹表达式 2›都只会执行一次，而‹符号›的类型会根据三重操作符（参见 2.3.16 节）的规则计算得到——这就是为什么会有操作符“?:”的原因，而它在运行时是不起作用的。经“?:”调整后的精细类型能保证，有符号与无符号类型之间的冲突，以及不同大小和精度数字之间可能潜在的冲突，都可以得到预防或者至少要少暴露出来。

还有一点值得注意，即编译器并不会为__n 强制分配一个特定的类型。因此，此种形式的 foreach 也可用于自定义类型，前提就是该类型要定义小于比较运算和递增运算（在第 12 章会了解到怎么做）。更妙的是，如果该类型没有定义小于比较运算，但定义了相等运算，那么编译器会在“降级”过程中将“<”替换成“!=”。在这种情形里，不会检查范围的有效性，因此你必须要保证从下限开始不断地执行“++”运算一定能到达上限；否则，迭代会出错[①]。

在‹符号›里可以指定一个实际的类型。该类型通常是多余的，只会在一些特殊情况下才会有用，例如，想确信迭代类型与期望类型是否相符合、解决有符号/无符号之间的歧义或者插入隐式转换：

```
import std.math, std.stdio;

void main() {
    foreach (float elem; 1.0 .. 100.0) {
        writeln(log(elem)); // 单精度对数
    }
    foreach (double elem; 1.0 .. 100.0) {
```

① 由于可用于多种类型，相等（不等）变得相当普遍，在这种情形下，C++语言的 STL 仍然坚持使用“!=”来测试迭代的结束条件。而 D 语言的方法则没那么普遍，在“<”可用的时候，巧妙地使用了它，其目的就是想在提高迭代的安全性的同时，在抽象或效率方面总体上不会有什么损失。

自在读书 www.zizidiary.com

```
        writeln(log(elem)); // 双精度对数
    }
    foreach (elem; 1.0 .. 100.0) {
        writeln(log(elem)); // 同上
    }
}
```

3.7.5　数组 foreach

继续看 foreach 的不同形式，这里的形式用于处理数组和分割：

```
foreach (‹符号›; ‹表达式›) ‹语句›
```

‹表达式›必须为数组（线性的或关联的）、分割或者自定义类型。对于最后一种情况，在第 12 章会进行讲解，因此这里的重点是数组和分割。‹表达式›计算过一次之后，在私有的临时变量里会存储一个对它的引用（实际的数组并不会被复制）。然后‹符号›会被依次绑定到该数组的每一个元素上，并执行‹语句›。与范围 foreach 一样，可以在‹符号›的前面指定一个类型。

foreach 语句假定数组的长度在迭代期间不会发生变化。如果会发生变化，则需要使用普通的 for 循环以及合适的循环总量。

迭代期间更新　在‹语句›内部对‹符号›进行赋值并不会反映到数组中。如果想要更改正被迭代那个元素，则需要使用前缀 ref 或 ref ‹类型›，于是‹符号›被定义成一个引用类型。例如：

```
void scale(float[] array, float s) {
    foreach (ref e; array) {
        e *= s; // 就地更新数组
    }
}
```

可以完整地指定一个带有 ref 的类型，例如在上面代码中，可以使用 ref float e 这样的形式。不过，这时的类型必须要能完全匹配。带 ref 时无法进行转换！

```
float[] arr = [ 1.0, 2.5, 4.0 ];
foreach (ref float elem; arr) {
    elem *= 2; // 很好
}
foreach (ref double elem; arr) { // 出错!
    elem /= 2;
}
```

原因很简单：为保证正确的赋值，ref 依赖于表示精确匹配。虽然可以在任何时候根据一个 float 来创建一个 double，但考虑到种种因素（最简单的就是它们拥有不同的大小），无法使用 double 赋值来更新 float。

我在哪里　有时，需要访问选代索引值。可以使用下面的形式来绑定一个符号：

```
foreach (‹符号 1›, ‹符号 2›; ‹表达式›) ‹语句›
```

因此可以写成：

```
void print(int[] array) {
    foreach (i, e; array) {
        writefln("array[%s] = %s;", i, e);
    }
}
```

自在读书 www.zizidiary.com

此函数会输出数组的内容，它与实际的 D 语言代码 print([5, 2, 8])产生的结果一样：

```
array[0] = 5;
array[1] = 2;
array[2] = 8;
```

访问关联数组的迭代索引值则更有意思：

```
void print(double[string] map) {
   foreach (i, e; map) {
      writefln("array['%s'] = %s;", i, e);
   }
}
```

现在调用 print(["Moon": 1.283, "Sun": 499.307, "Proxima Centauri": 133814298.759])，输出结果为：

```
array['Proxima Centauri'] = 1.33814e+08;
array['Sun`] = 499.307;
array[`Moon`] = 1.283;
```

请注意，元素的顺序与字面量里指定的顺序并不相同。事实上，在不同编译器的实现里或者同一编译器实现的不同版本里，测试同一段代码，得到的顺序也可能不同。这是因为关联数组为实现元素的高效存储和获取使用了几项深奥的技术，其代价就是无法保证顺序。

索引值的类型和元素的类型都会被推断。另外，也可以通过为‹符号$_1$›和‹符号$_2$›指定类型来强制它们的类型。不过‹符号$_1$›绝不能是 ref 类型。

技巧　在迭代期间，随意更改底层数组可能会产生下列影响。

- *数组就地突变*：迭代能“看到”对数组里仍未访问过的槽上的突变。
- *数组长度发生变化*：迭代会一直持续到在进入循环时数组所拥有的长度。数组大小的变化有可能会将新数组移动到内存的不同区块。这时，数组随后的突变对于此迭代是不可见的，而由此迭代自己所引发的后续突变对于数组也是不可见的。不推荐此操作，因为触发数组移动的规则与编译器具体实现有关。
- *使用底层分配控制函数，实现数组就地释放或收缩*：想获得完全控制和高效率，则需要花点时间阅读一下编译器实现文档里关于底层分配控制的内容。这里只能假定你清楚自己在做什么，并且假定对所有还未编写过垃圾回收器的人来说你是个比较乏味的人。

3.7.6　continue 语句和 break 语句

continue 语句会强制跳转到由最里面的 while、do-while、for 和 foreach 语句控制的那些语句的末尾。此循环会跳过其余由它控制的语句和进程而进入下一轮迭代。

break 语句会强制跳转到由最里面的 while、do-while、for、foreach、switch 和 final switch 语句控制的那些语句之后，实际上是结束执行。

两条语句都可以支持标签，即允许指定哪一条确切的语句生效。带标签的 break 和 continue 大大简化了复杂迭代模式的表达式，不需要状态变量，也不需要求助于你已知的那一条语句。这一点在下一节进行描述。

自在读书 www.zizidiary.com

```
void fun(string[] strings) {
   loop: foreach (s; strings) {
      switch (s) {
          default: ...; break;    // 跳出 switch
          case "ls":...; break; // 跳出 switch
          case "rm":...; break; // 跳出 switch
          ...
          case "#": break loop;   // 忽略掉其余串（跳出 foreach）
      }
   }
   ...
}
```

3.8 goto 语句

随着全球变暖，对围绕 goto 的争论火上浇油已没任何意义。我只想说，D 语言提供的形式如下：

```
goto ‹标签›;
```

这里的符号‹标签›在调用 goto 的函数里必须是可见的。在语句前面添加一个带有冒号的符号可以显式定义一个标签。例如：

```
int a;
...
mylabel: a = 1;
...
if (a == 0) goto mylabel;
```

在同一个函数里，不能重复定义标签。另一个限制是，如果一个值在跳入点可见，那么 goto 不能跳过这个值的定义点，即使该值未使用也一样不行。例如：

```
void main() {
  goto target;
  int x = 10;
  target: {} // 出错！goto 跳过了 x 的定义！
}
```

最后一点，goto 不能跨越异常的边界。关于这个限制 3.11 节会进行解释。除此以外，goto 几乎不受任何限制，这也恰恰让它变得非常危险。goto 能跳转的地方有：向前或向后、进出 if 语句、进出循环（其中包括臭名昭著的向前跳转至一个循环的中间）。

不过，在 D 语言里，不是所有与 goto 名字有关的事情都是危险的。在 switch 语句里，下面的语句，可以跳转到相应的 case ‹表达式›标签处：

```
goto case ‹表达式›;
```

而下面的语句则是跳转到 default 标签处：

```
goto default;
```

虽然这些跳转并不比其他 goto 更结构化，但它们更易跟踪，这是因为它们都是本地化的，而且大大简化了 switch 语句：

```
enum Pref { superFast, veryFast, fast, accurate,
   regular, slow, slower };
Pref preference;
double coarseness = 1;
...
switch (preference) {
```

自在读书 www.zizidiary.com

```
    case Pref.fast: ...; break;
    case Pref.veryFast: coarseness = 1.5; goto case Pref.fast;
    case Pref.superFast: coarseness = 3; goto case Pref.fast;
    case Pref.accurate: ...; break;
    case Pref.regular: goto default;
    default: ...
    ...
}
```

带有标签的 break 和 continue（参见 3.7.6 节）、异常（参见 3.11 节）以及高级流控制语句 scope（参见 3.13 节），像这样一些技术，很难找到令人信服的理由让 goto 的狂热追随者来使用它们。

3.9 with 语句

Pascal 语言引入的 with 语句可以让一个特定对象的处理变得很舒适。此语句的格式为：

```
with (‹表达式›) ‹语句›
```

它首先会计算‹表达式›，接着让结果对象的成员在‹语句›内部成为顶层可见的符号。在第 1 章我们已经遇到过结构（struct），现在一起来看一个包含 struct 的简短示例：

```
import std.math, std.stdio;

struct Point {
    double x, y;
    double norm() { return sqrt(x * x + y * y); }
}

void main() {
    Point p;
    int z;
    with (p) {
        x = 3;    // 对 p.x 赋值
        p.y = 4; // 显式使用 p 也没问题
        writeln(norm()); // 输出 p.norm，即为 5
        z = 1;           // z 仍然可见
    }
}
```

对这些域的更改直接反映到 with 所操作的那个对象上去了——with 会辨别并维持 p 是一个左值这样一个事实。

如果通过 with 变为可见的符号会掩盖函数之前定义的符号，那么编译器会拒绝访问，因为这里存在歧义。即使 Point 的定义是一样的，下面的代码也无法编译：

```
void fun() {
    Point p;
    string y = "I'm here to make a point (and a pun).";
    with (p) {
        writeln(x, ":", y); // 出错!
        // 不允许使用 p.y 将 y 遮掩掉!
    }
}
```

不过，这种错误总是在实际出现歧义时进行提醒，对于可能存在歧义的情形则不会提醒。

自在读书 www.zizidiary.com

例如，如果上面的 with 语句根本就不使用 y，那么这段代码也能进行编译并运行，尽管它可能存在歧义。当然，如果将 writeln(x, ":", y) 替换成 writeln(x, ":", p.y) 也不会有问题，因为 y 的显式修饰操作能消除任何可能的歧义。

with 语句可以掩盖掉模块级（如全局）的符号。使用 with 语句访问这些被掩盖的符号可以使用的“.符号”这样的语法。

请注意，使用下面的形式可以实现让多个成员变成隐式的：

```
with (<表达式1>) with (<表达式2>) ... with (<表达式n>) <语句>
```

使用嵌套的 with 不存在与歧义相关的危险，这是因为 D 语言不允许内层 with 引入的符号去掩盖外层 with 引入的符号。简言之，在 D 语言里，一个局部符号绝不会掩盖掉另一个局部符号。

3.10 return 语句

想要立即从当前函数返回一个值，可以编写下面的语句：

```
return <表达式>;
```

此语句会计算<表达式>的值，然后将它返回给调用者，同时将它隐式转换成（如有必要）函数的返回类型。

如果当前函数的类型为 void，那么<表达式>要么被省略，要么由返回类型依次为 void 的调用构成。

执行流在退出一个非 void 函数时，如果没有 return 语句，则是非法的。在编译期间很难高效地强制这一点，因此在极少情形里，编译器可能出现不必要的报错。

3.11 throw 语句和 try 语句

D 语言可以通过异常来支持错误处理。一个异常是由 throw 语句进行初始化的，并通过 try 语句进行处理。抛出异常的典型写法是：

```
throw new SomeException("Something fishy happened");
```

类型 SomeException 必须要继承内建的类 Throwable。D 语言不支持抛出任意类型，部分原因是它选择一个特定的根类型，这样已能让不同类型的异常很容易地链接在一起（稍后就会看这一点）。

想要处理一个异常或者最低限度地响应它，可以使用 try 语句。一般情况下，它看起来像下面这个样子：

```
try <语句>
catch (<E1> <e1>) <语句1>
catch (<E2> <e2>) <语句2>
...
catch (<En> <en>) <语句n>
finally <语句f>
```

这里的 finally 子句，以及所有的 catch 子句都可以省略，不过，try 语句必须至少要

自在读书 www.zizidiary.com

有一个 catch 子句或者 finally 子句。‹E_k›是类型，如前所述，它必须继承 Throwable。符号名‹e_k›会绑定到被捕捉到的异常对象上，也是可以省略的。

其语义如下所述。首先，它会执行‹语句›。如果它抛出了异常（假设为‹e_x›，其类型为‹E_x›），那么就试探性地将‹E_x›依次与类型‹E_1›、‹E_2›、……、‹E_n›进行匹配。第一个匹配到的类型‹E_k›，它要么就是‹E_x›，要么是它的父类。符号‹e_k›会绑定到异常对象‹e_x›，然后执行‹语句$_k$›。这样就可以认为这个异常已被处理完成。这时，如果‹语句$_k$›自己又抛出一个异常，则该异常并不会在当前异常的 catch 块里进行匹配。如果所有类型‹E_k›都未匹配成功，那么‹e_x›会到上层调用栈继续查找合适处理器。

finally 子句‹语句$_f$›，如果存在，则保证会在所有情况下都得到执行：不论是有没有异常抛出的情况，还是像当某个 catch 子句捕捉到了异常，然后又抛出了另外一个异常这样的情况。它实质上是保证能执行到的（当然，除非有无限循环或系统调用导致程序中止）。如果‹语句$_f$›抛出了一个异常‹e_f›，那么该异常会被添加到当前的异常链里。第 9 章包含了关于 D 语言的异常模型的完整信息。

goto 语句（参见 3.8 节）不能跳转到‹语句›，‹语句$_1$›，…，‹语句$_n$›以及‹语句$_f$›当中，除非 goto 也处于该语句中。

3.12　mixin 语句

在第 2 章（详见 2.3.4.2 节）我们已经知道 mixin 表达式允许将编译期间已知的字符串转换成 D 语言表达式，而且它们可以像普通代码一样被编译。mixin 语句则更进一步——不仅可以用 mixin 生成表达式，也可以用其生成声明和语句。

例如，假设要非常快速地计算出在一个字节里非零位（二进制位）的个数。这种统计即是众所周知的汉明权重（Hamming weight），它在很多应用程序中都有用处，如加密、分布式计算以及准数据库搜索。最简单的统计字节里非零位个数的方法是在依次不断累加最低有效位的同时对输入数据进行移位。还有一种更快的方法，它是由 Peter Wegner[60]首先提出的，然后被 Kernighan 和 Ritchie 在他们的 K&R 经典著作[34]里得到推广：

```
uint bitsSet(uint value) {
   uint result;
   for (; value; ++result) {
      value &= value - 1;
   }
   return result;
}
unittest {
   assert(bitsSet(10) == 2);
   assert(bitsSet(0) == 0);
   assert(bitsSet(255) == 8);
}
```

这种方法比那种采用简单处理的方法更快，其原因在于它执行的迭代次数恰好与设置的位数一样多。不过 bitsSet 仍然拥有控制流，更快速的方法是只进行表查询。为获得最佳效果，我们需要在编译期间填充该表，这正是 mixin 声明可以派上大用场的地方。具体计划是，首先生成一个字符串，它看起来很像是声明一张表，然后使用 mixin 将该字符串编译成常规的代码。表生成器有点像下面这个样子：

自在读书 www.zizidiary.com

```
import std.conv;

string makeHammingWeightsTable(string name, uint max = 255) {
   string result = "immutable ubyte["~to!string(max + 1) ~"] "
      ~name~" = [ ";
   foreach (b; 0 .. max + 1) {
      result ~= to!string(bitsSet(b)) ~ ", ";
   }
   return result ~ "];";
}
```

调用makeHammingWeightsTable("t")会返回一个字符串"immutable ubyte[256] t = [0, 1, 1, 2, ……, 7, 7, 8,];"。其中的修饰符immutable（参见第8章）说明该表在初始化之后绝不会被更改。我们首先看到的是库函数to!string（参见1.5节），它会将所有数据（在这里，是bitsSet返回的uint）转换成一个字符串。一旦以字符串格式获得所需要的编码，那么距离表的定义也就只有一步之遥：

```
mixin(makeHammingWeightsTable("hwTable"));
unittest {
   assert(hwTable[10] == 2);
   assert(hwTable[0] == 0);
   assert(hwTable[255] == 8);
}
```

你可以构建几乎任何大小的表格，不过还是建议多做测试——太大的表格的查询实际上可能会因为受到高速缓存的影响而比通过计算实现的查询更慢。

最后，像合气道教练，他们是不愿意在课堂上推荐防狼喷雾剂的，但我们还是得提一下——将字符串import（参见2.2.5.1节）与混合声明mixin相结合，可以提供模块化的最底层形式：文本包含。例如：

```
mixin(import("widget.d"));
```

import表达式会读入文件widget.d里的文本，并将它当做一个字面量字符串，接着mixin表达式会立即将它转换成代码。在你确信通往黑客荣耀之路就依赖于这一技巧之后，就可以用它了。

3.13 scope语句

尽管scope语句的功能也可以在其他语言里找到（或许略有差异），但它仍然是D语言的一个创新。在D语言里，使用scope很容易就能编写出异常安全（exception-safe）的代码。更为重要的是，在以后还能很好地阅读和理解它。使用scope的代码所实现的功能也可以借助其他方式来完成，不过除了一些最普通的例子外，其结果都是令人费解的。

```
scope(exit) ‹语句›
```

这条语句会植入‹语句›，当控制流离开当前作用域时它才会被执行。它跟try语句的finally子句所完成的动作是一样的，但它具有更好的可伸缩性。想确保在离开一个作用域时“记得”要以良好的顺序进行，使用scope(exit)则能带来方便。例如，有一个应用程序，它带有一个“verbose”标志，想要临时禁用时，则可以写成这样：

自在读书 www.zizidiary.com

```
bool g_verbose;
...
void silentFunction() {
    auto oldVerbose = g_verbose;
    scope(exit) g_verbose = oldVerbose;
    g_verbose = false;
     ...
}
```

silentFunction 的其余部分可以包含任意代码，其中包括带有提前返回和可能引发异常的代码。可以满怀信心地说，无论遇到什么状况，当 silentFunction 执行结束时，g_verbose 都会被正确地还原。

推广到更一般的情况，我们为 scope(exit)定义一个“降级”的概念，即一种将使用了 scope(exit)的代码重写为包含其他语句（如 try）的等价代码的系统性的方法。在利用 while 语句对 for 语句进行解释的时候，以及利用 for 语句对 foreach 语句进行解释的时候，我们都已非正式地使用了“降级”。

假设有一块包含 scope(exit)的语句：

```
{
    ‹语句1›
    scope(exit) ‹语句2›
    ‹语句3›
}
```

先选取代码块里的第一个 scope，因此，我们可以假定‹语句1›本身并未包含 scope（但‹语句2›和‹语句3›则有可能）。降级会将此段代码转换成下面这个样子：

```
{
    ‹语句1›
    try {
        ‹语句3›
    } finally {
        ‹语句2›
    }
}
```

随着转换，‹语句3›和‹语句2›会被进一步降级，因为它们也有可能会包含其他的 scope 语句。因为与最初的序列相比，分段总是严格更小的，所以降级一定总会结束。这就意味着，包含多重 scope(exit)的语句是定义良好的，即使像 scope(exit) scope(exit) scope(exit) writeln("?")这种怪异的情形也一样。特别地，一起来看看在同一个代码块里存在两个 scope(exit)语句这种有意思的情形里到底发生了什么：

```
{
    ‹语句1›
    scope(exit) ‹语句2›
    ‹语句3›
    scope(exit) ‹语句4›
    ‹语句5›
}
```

假设所有语句都不包含其他的 scope(exit)语句。在降级之后，可以得到：

```
{
    ‹语句1›
    try {
        ‹语句3›
```

自在读书 www.zizidiary.com

```
        try {
            ‹语句5›
        } finally {
            ‹语句4›
        }
    } finally {
        ‹语句2›
    }
}
```

展示这段笨拙代码的目的是要弄清楚，在同一代码块里的多条 scope(exit) 语句的执行顺序是什么。跟踪这个流程，最后结果表明‹语句4›是在‹语句2›之前执行的。一般情况下，scope(exit) 语句会以栈的 LIFO（后进先出）方式执行，即在执行流里反转它们的顺序。

跟踪 scope 版本的流程比跟踪等效的 try/finally 代码的流程要容易很多。简单地到达 scope 语句也就保证了在退出该作用域时它所控制的语句能被执行。这样，在你的代码中，不需要通过笨拙地嵌套 try/finally 语句，只需简单地以直线方式恰当地排列好语句，即可实现异常安全。

上一个示例也展示了 scope 语句的一种非常优秀的特性：可伸缩性。就其强大的可伸缩性而言，scope 是最耀眼的。要知道，如果我们需要的只是一个特殊场合才会用到的 scope，那么我们已经有了可通过手工编写与其近似等效功能代码的语言方式。当其自身也使用 scope(exit) 时，完成多条 scope(exit) 语句的功能会出现代码长度随之呈线性增长的情况；而使用与之等同的基于 try 的代码时，代码的长度和深度都会随之线性增长。除了与其他像 if 或 for 那样的复合语句能共享“不动产”外，大部分情况下深度的伸缩性都很差。C++语言式的析构函数（D 语言也支持，参见第 7 章）也提供了一种可伸缩的解决方案，只要能忽视定义新类型的成本；但如果主要是因为其析构函数（似曾感觉还需要一个像 CleanerUpper 一样的类？），而不得不定义一个类，那么这种可伸缩性就会比嵌入 try 语句所带来的可伸缩性还要糟糕。简言之，如果类是真空焊接，try/finally 是用于咀嚼的口香糖，那么 scope(exit) 便是一种速干混合胶。

scope(success) ‹语句›仅在当前作用域将完全正常退出（不是通过异常）时，才会安排‹语句›的执行。scope(success) 的降级过程如下：

```
{
    ‹语句1›
    scope(success) ‹语句2›
    ‹语句3›
}
```

变成了

```
{
    ‹语句1›
    bool __succeeded = true;
    try {
        ‹语句3›
    } catch(Exception e) {
        __succeeded = false;
        throw e;
    } finally {
        if (__succeeded) ‹语句2›
    }
}
```

自在读书 www.zizidiary.com

同样地，‹语句$_2$›和‹语句$_3$›一定会进一步降级，一直到它们不再包含 scope 语句为止。

继续往前推进到一个让人感觉抑郁的形式，执行 scope(failure) ‹语句›能够保证仅当当前作用域因异常退出时‹语句›才会被执行。

scope(failure)的降级几乎与 scope(success)的一样——它仅仅是将对 __succeeded 的测试变反而已。降级过程的变化为从

```
{
    ‹语句1›
    scope(failure) ‹语句2›
    ‹语句3›
}
```

变成了

```
{
    ‹语句1›
    bool __succeeded = true;
    try {
        ‹语句3›
    } catch(Exception e) {
        __succeeded = false;
        throw e;
    } finally {
        if (!__succeeded) ‹语句2›
    }
}
```

其中依然包含了对‹语句$_2$›和‹语句$_3$›的进一步降级。

在很多地方都可以放置 scope 语句进行应用。例如，要使用事务的方式来创建文件，也就是说，如果文件创建失败，就不要在磁盘上留有不完整的文件。对于此种需求可以这样来实现：

```
import std.exception, std.file, std.stdio;

void transactionalCreate(string filename) {
    string tempFilename = filename ~ ".fragment";
    scope(success) {
        std.file.rename(tempFilename, filename);
    }
    auto f = File(tempFilename, "w");
    // 随意往 f 里写入内容
}
```

scope(success)用于提前设置函数的目标。无 scope 代码的同等功能会更复杂一些，而且更为实际的是，大部分程序员都过多忙于让好的事情能在正常情况下工作，完全无法通过增加额外的工作来支持那些不太可能的情况。这也正是 D 语言应该将错误处理变得尽可能简单的原因。

这种编程风格能带来的一个好处就是，transactionalCreate 的开始部分集中了所有的错误处理代码，而它对主代码并未产生任何影响。与它所表示的内容一样简单，transactionalCreate 是绝对可靠的：在离开时，要么会有一个完好的文件，要么是一个破碎文件，但不会出现一个声明是正确但其实已损坏的文件。

3.14　synchronized 语句

synchronized 语句的形式如下：

自在读书 www.zizidiary.com

synchronized (‹表达式$_1$›, ‹表达式$_2$›...) ‹表达式›

synchronized 会影响多线程程序里的“区域锁定”（scoped locking）。第 13 章中将会定义 synchronized 的语义。

3.15　asm 语句

D 语言不支持一些汇编语言的接口，就无法遵守其成为一门系统级编程语言的承诺。因此，如果你喜欢自己动手，你会很高兴地听到 D 语言有一套针对 Intel x86 平台的经过精心定义的嵌入式的汇编语言。再好不过的是，D 语言的 x86 汇编语言是可移植的，它几乎可以跨越所有工作在 x86 机器上的 D 语言实现。让汇编语言只依赖于机器，而不是操作系统，这个特性听起来可能有些难以置信，你定会为之惊讶。出于历史原因，每一个操作系统都定义了自己的无法兼容的汇编语言语法，例如，你没办法让任何 Windows 里的汇编代码在 Linux 平台上也可以工作，这是因为它们的语法完全不一样（毫无道理可讲）。D 语言解决此难题的做法就是不依赖于系统相关的外部汇编。而事实上，编译器能够真正地解释和理解汇编语言语句。想要编写汇编代码，只需要采用下面形式即可：

asm ‹asm-语句›

或者

asm { ‹asm-语句› }

在 asm 可访问之前，位于 asm 块内部的符号是正常可见的，这保证了汇编代码能够访问 D 语言的实体。本书并没有全面讲述 D 语言的汇编语言，对于任何使用过 x86 汇编的人来说，它看起来应该是相当熟悉的。想要了解完整的细节，可以查阅 D 语言的汇编语言文档[12]。

3.16　小结和快速参考

D 语言在语句方面提供了好些亮点，其中包括不少很有意思的新内容，如 static if、final switch 和 scope。表 3-1 是关于 D 语言所有语句的快速参考，它在保持简洁的同时在极度精确和详尽细节方面（对于这些内容，可以参考本章相应的小节）作了一些牺牲。

表 3-1　语句速查表（‹*s*›代表语句、‹*e*›代表表达式、‹*d*›代表声明、‹*x*›代表符号）

语句	描述
‹*e*›;	计算‹*e*›的值。不起作用的表达式以及只涉及内建类型和操作符的表达式，则是静态禁止的（参见 3.1 节）
{‹s_1›……‹s_n›}	按词法顺序，依次执行‹s_1›到‹s_n›之间的语句，除非出现显式的转移操作（如通过 return）（参见 3.2 节）
asm ‹*s*›	依赖于机器的汇编代码（这里的‹*s*›是汇编代码，而非 D 语言语句）。目前只部分移植了 x86 汇编的相同语法，在所有支持的操作系统上都可使用（参见 3.15 节）
break;	通过将控制转移到正好位于相应控制语句之后的方式，停止当前的（最内层）switch、for、foreach、while 以及 do 语句（参见 3.7.6 节）

自在读书 www.zizidiary.com

续表

语句	描述
break ‹x›;	通过将控制转移到正好位于相应控制语句之后的方式，停止前面直接带有标签‹x›:的 switch、for、foreach、while 以及 do 语句（参见 3.7.6 节）
continue;	通过跳过相应控制语句余下部分的方式，继续执行当前的（最内层）for、foreach、while 以及 do 语句（参见 3.7.6 节）
continue ‹x›;	通过跳过相应控制语句余下部分的方式，继续执行前面直接带有标签‹x›:的 for、foreach、while 以及 do 语句（参见 3.7.6 节）
do‹s›while (‹e›);	先执行一次‹s›，然后在‹e›为非零值的前提下继续执行（参见 3.7.2 节）
for (‹s_1› ‹e_1›; ‹e_2›) ‹s_2›	先执行‹s_1›（它可以是表达式语句、值定义或者分号），然后在‹e_1›为非零值的前提下继续执行‹s_2›以及计算‹e_2›（参见 3.7.3 节）
foreach(‹x›;‹e_1›.. ‹e_2›) ‹s›	将‹x›初始化成‹e_1›，然后依次递增 1 个步长，只要‹x›<‹e_2›，便执行‹s›。如果‹e_1›>=‹e_2›，则不执行。‹e_1›和‹e_2›都只会被计算一次（参见 3.7.4 节）
foreach (ref$_{opt}$‹x›; ‹e›) ‹s›	通过声明‹x›，并将它依次绑定到‹e›的每一个元素的方式来执行‹s›。表达式‹e›的计算结果必须为一个数组类型或者自定义范围类型（参见 12 章）。如果存在 ref，则对‹x›的更改也会反映到被迭代的实例上（参见 3.7.5 节）
foreach(‹x_1›,ref$_{opt}$ ‹x_2›; ‹e›) ‹s›	与上面的用法相似，只不过它引入了额外的值‹x_1›。如果‹e›是一个关联数组，则‹x_1›会绑定到它的键上，而‹x_2›会绑定到其值上。否则，这个值会绑定到一个整数，用于统计此循环执行的次数（从零开始）（参见 3.7.5 节）
goto ‹x›;	将控制转移到标签‹x›处，在当前函数里必须要定义‹x›:（参见 3.8 节）
goto case ‹x›;	将控制转移到当前 switch 的 case 标签‹x›处（参见 3.8 节）
goto default;	将控制转移到当前 switch 的 default 标签处（参见 3.8 节）
if (‹e›) ‹s›	如果‹e›为非零值，则执行‹s›（参见 3.3 节）
if(‹e›) ‹s_1› else ‹s_2›	如果‹e›为非零值，则执行‹s_1›，否则执行‹s_2›。结尾的 else 会绑定到最后一个 if 或 static if（参见 3.3 节）
static if (‹e›) ‹d/s›	在编译期间计算‹e›的值，如果‹e›为非零值，则编译声明或语句‹d/s›。如果存在有{和}的话，则围绕‹d/s›的第一层括号会被剥掉（参见 3.4 节）
static if (‹e›) ‹d/s_1› else‹d/s_2›	与 static if 类似，只是多了一个 else 子句。结尾的 else 会绑定到最后一个 if 或 static if（参见 3.4 节）
return ‹e›$_{opt}$;	从当前函数返回。返回的表达式必须能转换成函数声明的返回类型。如果函数的返回类型为 void，则表达式‹e›可以省略（参见 3.10 节）

自在读书 www.zizidiary.com

续表

语句	描述
scope(exit) ‹s›	在当前作用域以任何方式（如返回、穿越[①]或抛出异常）退出时，执行‹s›。对于各种情况（包括 failure 和 success）的多重 scope 语句，其执行顺序与定义它们的顺序相反（参见 3.13 节）
scope(failure) ‹s›	在当前作用域通过抛出异常方式退出时，执行‹s›（参见 3.13 节）
scope(success) ‹s›	在当前作用域通过正常方式（返回或穿越）退出时，执行‹s›（参见 3.13 节）
switch (‹e›) ‹s›	计算‹e›的值，并跳转至匹配到的 case 标签处（包含‹s›内部）（参见 3.5 节）
Final switch (‹e›) ‹s›	与 switch 类似，不过它是用于枚举值，并静态强制要求所有值都必须使用 case 标签进行处理（参见 3.6 节）
synchronized (‹e_1›, ‹e_2›...) ‹s›	在锁定由‹e_1›、‹e_2›等返回的对象的同时，执行‹s›（参见 3.14 节）
throw ‹e›;	计算‹e›的值，并将它抛出，以便将控制转移到最近匹配到的那个 catch 处理器所在的位置。‹e›的类型必须为 Throwable 或者由它派生的类（参见 3.11 节）
try ‹s› catch(‹T_1› ‹x_1›) ‹s_1› ... catch(‹T_n› ‹x_n›) ‹s_n› finally ‹s_f›	执行‹s›。如果‹s›抛出了一个异常，则会尝试依次去匹配‹T_1›到‹T_n›这些类。如果找到匹配 k，则停止匹配，接着执行‹s_k›。对于所有情形，都会恰在将控制转移出 try 语句时（不管是正常地还是通过异常），执行‹s_f›。所有的 catch 子句或 finally 子句都可以省略，但不能两者都省略（参见 3.11 节）
while (‹e›) ‹s›	只要‹e›为非零值，就执行‹s›（如果在第一次计算时，‹e›为零，则不执行）（参见 3.7.1 节）
with (‹e›) ‹s›	计算‹e›的值，然后执行‹s›，它就像是‹e›的类型的成员函数，即在‹s›里使用的所有符号都会优先查找‹e›的成员（参见 3.9 节）

① 英文原词为 fall through。它指的是，在 switch 语句中，执行流程在一种情况结束时未使用 break 语句进行跳转，而是顺序进入到下一种情况继续执行。D 语言支持这种机制，而有的语言（如 C#、Pascal 等）则不允许。——译者注

自在读书 www.zizidiary.com

第 4 章

数组、关联数组和字符串

在前几章，我间接了解了数组、关联数组以及字符串——其中有表达式，也有字面量，因此现在就来深入探讨一下。很多优秀的代码都可以只使用这 3 种类型来编写，因此，既然我们已经讲过了表达式和语句，接下来就学习一下这 3 种类型。

4.1 动态数组

D 语言提供了一个简单但非常通用的数组抽象类型。对于类型 T，T[]指的是这样一种类型，它会占用一段连续的内存，其中还包含类型为 T 的元素。D 语言将 T[]称作“类型为 T 的值的数组”或者以口语化的形式将其称作“T 类型数组”。

想要创建动态数组，可以像下面这样使用 new 表达式（参见 2.3.6.1 节）：

```
int[] array = new int[20]; // 创建一个由 20 个整数构成的数组
```

或者使用更为简单和方便的形式：

```
auto array = new int[20]; // 创建一个由 20 个整数构成的数组
```

新创建的类型为 T[]的数组，其所有元素都会被初始化为 T.init（对于整数，它为 0）。创建之后，数组的元素可通过索引表达式 array[n]来进行访问：

```
auto array = new int[20];
auto x = array[5];              // 有效索引值为 0 到 19
assert(x == 0);                 // 元素的初始值为 int.init，即 0
array[7] = 42;                  // 所有元素都是可赋值的
assert(array[7] == 42);
```

传递到 new 表达式的那个元素个数不要求一定是常量。例如，下面的程序就能创建一个随机长度的数组，并使用随机数来填充它。为产生随机数，需要借助于模块 std.random 里的函数 uniform：

```
import std.random;

void main() {
```

自在读书 www.zizidiary.com

```
    // 拥有 1 到 127 之间任意个数的元素
    auto array = new double[uniform(1, 128)];
    foreach (i;  0 .. array.length) {
        array[i] = uniform(0.0, 1.0);
    }
    ...
}
```

上面的 foreach 循环可以不使用索引，而是改成直接引用每一个数组元素(参见 3.7.5 节)：

```
    foreach (ref element; array) {
        element = uniform(0.0, 1.0);
    }
```

ref 会告知编译器，我们想要的是将赋予 element 的值反射回原始数组。不然的话，element 会依次为每一个数组元素的副本。

如果想要将一个数组初始化成特定的内容，则可以使用数组字面量：

```
auto somePrimes = [ 2, 3, 5, 7, 11, 13, 17 ];
```

另一种创建数组的方式是复制已有的数组。特性 array.dup 会逐个元素地产生一个 array 的副本：

```
auto array = new int[100];
...
auto copy = array.dup;
assert(array !is copy);              // 这些数组是不同的
assert(array == copy);               //       但拥有相同的内容
```

最后，如果只定义了类型为 T[]的变量，而未进行初始化，或者是使用 null 进行初始化的，那么它就是一个空数组。一个空数组没有元素，并且可以与 null 进行相等性比较。

```
string[] a;          // 等同于 string[] a = null
assert(a is null);
assert(a == null);   // 同上
a = new string[2];
assert(a !is null);
a = a[0 .. 0];
assert(a !is null);
```

在上面的代码片段的最后一行提示了一个奇怪的细节，即一个空的数组不一定为 null。

4.1.1　长度

动态数组会记住它们的长度。要访问这一长度，可以使用数组的.length 特性：

```
auto array = new short[55];
assert(array.length == 55);
```

表达式 array.length 经常出现在 array 的索引表达式内部。例如，array 的最后一个元素为 array[array.length - 1]。为简化它，可以在索引表达式内部使用符号$来表示“正被索引数组的长度”。

```
auto array = new int[10];
array[9] = 42;
assert(array[$ - 1] == 42);
```

对数组长度的有效更改会在 4.1.8 节、4.1.9 节和 4.1.10 节进行讨论。

自在读书 www.zizidiary.com

4.1.2　边界检查

如果照下面这样做会发生什么呢？

```
auto array = new int[10];
auto invalid = array[100];
```

假定这些数组已经知道自己的长度，那么就可能插入适当的边界检查，因此可行性不是问题。唯一的问题在于边界检查是一个效率与安全性纠结不清的典型案例。

出于安全的考虑，不管通过何种方式，都有必要确保数组访问是在边界之内进行的。边界外的访问会出现无法确定的行为，并且使程序很容易遭到攻击和破坏。

不过，完全的边界检查在当前的编译器技术条件下仍然会对运行效率产生很大的影响。高效的边界检查是大家都在研究的目标。有一种比较流行的方法是开始时使用带完全检查的程序，然后将静态分析程序证明多余的检查移除。在一般情况下，这种做法很快就会变得难于处理，尤其是当数组在多个进程和模块之间都被用到的时候。如今的方法即使对于很普通的程序也需要花费很长一段分析时间，而且只能移除一小部分检查[58]。

在边界检查这个难题面前，D 语言处于一个有些矛盾的位置。D 语言在努力提供现代语言的安全性和便捷性的同时，还要提供系统级程序员所追求的终极高性能。边界检查问题暗示着要在两者之间进行选择，而 D 语言允许你进行选择，而不是为你做了选择。

D 语言在编译期间有两个特别突出的地方：

- 安全模块与系统模块（参见 11.3.1 节）；
- 非发行构建与发行构建（参见 10.6 节）。

D 语言将模块分成了“安全”（safe）模块与“系统”（system）模块两种。有一个处于中间的安全级别叫作“信任的”（trusted），即是说此模块对外展示的是安全接口，但其实现内部可能使用了系统级的访问。你可以决定每一个模块的分类。当编译一个安全模块时，编译器会静态禁用所有可能引起内存损坏的语言特征，其中包括不检查数组索引。当编译一个系统模块或信任模块时，编译器会允许原始的、未检查的硬件访问。你可以选择是否要将模块的指定部分标识成安全的、系统的或信任的，标识方法可以使用命令行选项，也可以插入一个属性，如下所示：

```
@safe:
```

或者

```
@trusted:
```

或者

```
@system:
```

从插入点开始，选定的安全级别就开始发挥作用，一直到使用了另一个安全级别或者直到文件末尾为止。

第 11 章会详细解释模块安全性是如何工作的，在这里只是透露一点重要信息，即对于应用程序开发人员，可以选择正处理的模块到底是`@safe`、`@trusted`还是`@system`。大部分情况下（当然不是全部），一个典型的应用程序的模块应该是`@safe`。

自在读书 www.zizidiary.com

模块的安全性与应用程序选择的发行构建方式是正交的。通过往 D 语言编译器传递一个命令行标志（在参考编译器实现 dmd 里为-release），可以让它构建一个发行版本。在安全模块里，数组边界总是会被检查。在系统模块里，边界检查只会插入到非发行构建里去。在非发行构建里，编译器也会插入像 assert 表达式和契约断言（关于发行模式细节的完整讨论请参见第 10 章）这样的一些检查。表 4-1 总结了安全模块对系统模块与发行模式对非发行模式之间的交互关系。

表 4-1　依赖于模块种类和构建模式的边界检查

	安全模块	系统模块
非发行构建	✔	✔
发行构建（在 dmd 里为-release）	✔	☠

已警告过你了。

4.1.3　分割

分割（slicing）是一项很强大的功能，它允许只选择和处理一段连续的数组内容。例如，只想要输出数组最后半截的内容：

```
import std.stdio;

void main() {
   auto array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
   // 只输出最后半截
   writeln(array[$ / 2 .. $]);
}
```

上面程序的输出为：

```
5, 6, 7, 8, 9
```

想要提取数组的一个分割，需要使用记号 array[m .. n]，此记号会从数组里提取部分内容，范围是从索引 m 开始，直到（包括）索引 n－1 结束。分割的类型与 array 一样，因此可以将分割重新赋予原来那个数组，举例如下：

```
array = array[$ / 2 .. $];
```

符号$可以出现在用于限制分割范围的表达式里——好比一个简单的索引值——它表示的是正被分割的那个数组的长度。对于像 m == n 这样的情形是可接受的，它会产生一个空的分割。不过，对于 m > n 或者 n > array.length 这样的分割，则是非法的。对这种非法情形的检查遵从前面（参见 4.1.2 节）描述过的边界检查规则。

表达式 array[0 .. $]会提取包含整个 array 内容的分割。由于此表达式会经常遇到，因此 D 语言索性就更进一步，让 array[]等同于[0 .. $]。

4.1.4　复制

一个数组对象最低限度也会保留（或者在几乎可以忽略的时间内计算）两条关键信息，即

自在读书 www.zizidiary.com

数据块边界的上限和下限。例如，下面的计算就会出现图 4-1 所展示的状态：

```
auto a = [1, 5, 2, 3, 6];
```

此数组只能“看到”其边界之间的区域。带灰网的区域对它来说是不可访问的。

也有可能是其他的表示，例如存储第一个元素的地址和块的长度，或者第一个元素的地址和恰好位于最后一个元素之后的地址。所有这些表示都能访问到相同的信息。

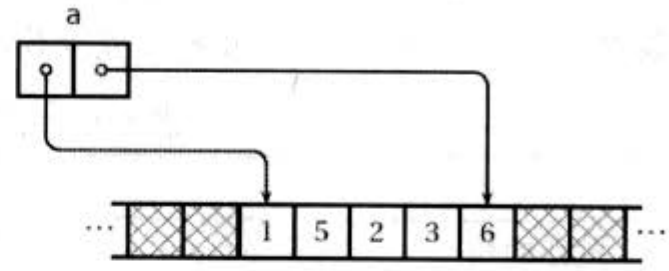


图 4-1　引用包含 5 个元素的区块的数组对象

通过另一个数组来初始化一个数组(如 auto b = a;)或者通过另一个数组来给一个数组赋值（如 int[] b; ... b = a;）在底层都不会自动复制数据。此操作只是简单地让 b 引用了与 a 一样的内存区块，如图 4-2 所示。

此外，从 b 里提取一个分割，并用来减小 b 所“看到的”数据块，也不会导致数据复制。从图 4-2 所示状态开始，如果我们现在执行下面的语句：

```
b = b[1 .. $ - 2];
```

那么 b 的范围就会收缩，但不会出现数据复制（参见图 4-3）。

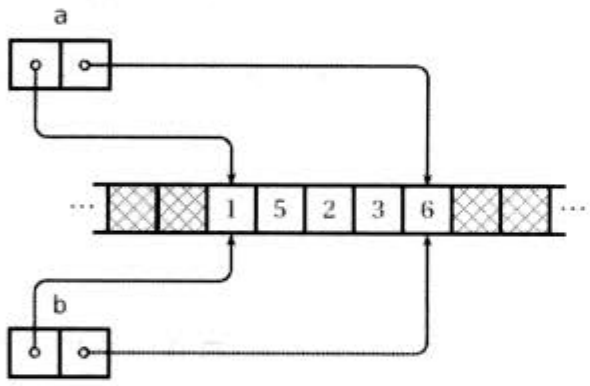


图 4-2　执行 auto b=a;并不会复制 a 的内容，而是会创建一个新的引用同一数据块的数组对象

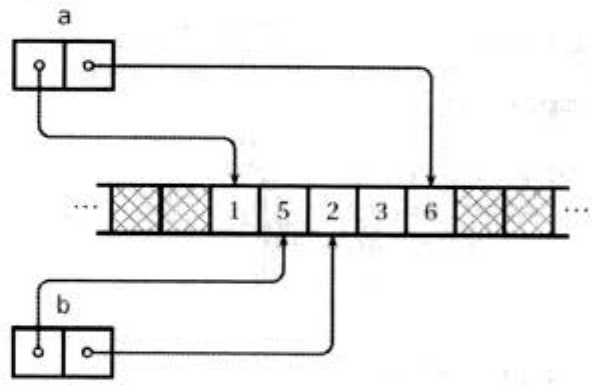


图 4-3　执行 b = b[1 .. $ - 2];会使 b 控制的数据块收缩，但不会复制所选择的分割

在这几张图里所展示的数据共享有一个很直接的后果，那就是往某个数组的元素写入内容也会反映到其他数组上：

```
int[] array = [0, 1, 2];
int[] subarray = array[1 .. $];
assert(subarray.length == 2);
subarray[1] = 33;
assert(array[2] == 33); // 对 subarray 的写入影响了 array
```

4.1.5　相等性比较

表达式 a is b（参见 2.3.4.3 节）会比较两个数组的边界的相等性，仅当 a 和 b 都绑定到内存里的同一区块时结果才为 true。根本就不会进行内容的比较。

想要对 a 和 b 按逐个元素进行相等比较，则可以使用 a == b 或者它的否定操作 a != b（参见 2.3.12 节）。

自在读书 www.zizidiary.com

```
auto a = ["hello", "world"];
auto b = a;
assert(a is b);                    // 通过，a 和 b 的边界相同
assert(a == b);                    // 通过，肯定相等
b = a.dup;
assert(a == b);                    // 通过，虽然位置不一样，但 a 和 b 还是相同
assert(a !is b);                   // 通过，虽然内容相同，但 a 和 b 不同
```

相等性比较会逐个迭代两个数组的所有元素，并依次使用==来进行比较。

4.1.6　拼接

拼接表达式的结构为：

```
lhs ~ rhs
```

拼接后的结构是一个新数组，其内容为 lhs 再紧跟 rhs 的内容。可以进行拼接的数据有：两个类型为 T[]和 T[]的数组，数组与值（T[]和 T），以及值与数组（T 和 T[]）。

```
int[] a = [0, 10, 20];
int[] b = a ~ 42;
assert(b == [0, 10, 20, 42]);
a = b ~ a ~ 15;
assert(a.length == 8);
```

拼接总是会分配一个新的数组。

4.1.7　数组表达式

有一些运算不需要显式地迭代便可以应用于整个数组。要创建一个数组表达式，方法是在表达式所涉及的（包括赋值表达式左边）所有分割上都指定一个[]或者[m .. n]作为结尾：

```
auto a = [ 0.5, -0.5, 1.5, 2 ];
auto b = [ 3.5, 5.5, 4.5, -1 ];
auto c = new double[4];                    // 必须先分配
c[] = (a[] + b[]) / 2;                     // 使用 a 和 b
assert(c == [ 2.0, 2.5, 3.0, 0.5 ]);
```

数组表达式的形式如下：

- 一个单值，如 5；
- 结尾显式带有[]或[m .. n]的分割，如 a[]或 a[1 .. $ - 1]；
- 所有有效的涉及以上两项的 D 语言表达式，一元操作符-和~，以及二元操作符+、-、*、/、%、^^、^、&、|、=、+=、-=、*=、/=、%=、^=、&=和|=。

数组表达式的效果就是依次对左边每一个元素进行循环赋值，赋值内容为右边与之对应索引位置上的值。例如，赋值

```
auto a = [1.0, 2.5, 3.6];
auto b = [4.5, 5.5, 1.4];
auto c = new double[3];
c[] += 4 * a[] + b[];
```

等同于

自在读书 www.zizidiary.com

```
foreach (i; 0 .. c.length) {
    c[i] += 4 * a[i] + b[i];
}
```

边界检查规则的应用正常情况下会根据 4.1.2 节所描述的内容进行。

利用后缀带在[]或'[m .. n]'的分割、数字以及允许的操作符，就可以组成任何深度和复杂度的括号表达式。例如：

```
double[] a, b, c;
double d;
 ...
a[] = -(b[] * (c[] + 4)) + c[] * d;
```

按数组运算的一个流行用法是完成简单的填充和复制：

```
int[] a = new int[128];
int[] b = new int[128];
 ...
b[] = -1;                    // 使用-1 来填充 b
a[] = b[];                   // 将 b 的数据复制到 a
```

警告 数组运算很强大，但能力越大责任越大。在数组运算中，你需要保证任何赋值操作的左值与右值这两个部分不会出现重叠的情况。编译器无法保证在什么时候能将这类操作优化成由主机处理器提供的基本向量操作。如果出现重叠，则需要通过手工编写这些循环，这时也不要让编译器做任何未经确认过的假设。

4.1.8　收缩

数组收缩（array shrinking）表示的是数组从左端或右端开始“忘记”一些元素，同时不需要移动其他元素。对移动的限制很重要：如果移动元素是可选项，则数组就可能很容易实现收缩——只需要创建一个新的副本，让它包含那些被保留的元素。

收缩数组是最轻松的事情：只需将数组的一个分割赋予自己即可。

```
auto array = [0, 2, 4, 6, 8, 10];
array = array[0 .. $ - 2];          // 右边收缩两个元素
assert(array == [0, 2, 4, 6]);
array = array[1 .. $];              // 左边收缩一个元素
assert(array == [2, 4, 6]);
array = array[1 .. $ - 1];          // 在两端进行收缩
assert(array == [4]);
```

所有收缩操作花费的时间与数组的长度无关（事实上，它们仅包含几个字的赋值）。支持两端收缩是 D 语言数组的一项非常有用的功能。（其他语言支持从右端进行低成本的数组收缩，但不支持从左边收缩，主要原因在于左边收缩为保持数组左边界的位置可能会移动数组中的所有元素。）在 D 语言里，为系统地处理数组的元素，可以利用该数组的副本渐步对它进行收缩；相信时间恒定的收缩操作对处理时间并没有显著影响。

举例来说，我们要编写一个小程序，它能够检测通过命令行传递的回文数组。一个回文数组自左向右是对称的，如[5, 17, 8, 17, 5]就是一个回文数组，但[5, 7, 8, 7]就不是。我们需要几个辅助操作。首先是命令行获取。如果将 main 函数定义为 main(string[] args)，则很容易就能以一个字符串数组的形式获得命令行的内容。接着，我们需要将参数从字符串转换成整

自在读书 www.zizidiary.com

数。为此，可以使用在模块 std.conv 里定义的合适的函数。对于字符串 str，通过计算 to!int(str)来将 str 解析成一个整数。利用这些功能，我们就可以编写回文测试程序了：

```
import std.conv, std.stdio;

int main(string[] args) {
   // 去掉程序名称
   args = args[1 .. $];
   while (args.length >= 2) {
      if (to!int(args[0]) != to!int(args[$ - 1])) {
         writeln("not palindrome");
         return 1;
      }
      args = args[1 .. $ - 1];
   }
   writeln("palindrome");
   return 0;
}
```

首先，程序必须从参数列表里去掉程序名（遵从 C 语言所建立的传统）。当像下面那样调用这个程序(将其称作 palindrome)时，数组 args 就会包含["palindrome", "34", "95", "548"]这样一些内容：

```
palindrome 34 95 548
```

在这里，从左边进行的收缩 args = args[1 .. $]就能派上用场了，它会将参数减少为["34", "95", "548"]。然后程序迭代地比较数组的两端。如果它们不相等，则没有必要继续进行测试，因此输出"not palindrome"，接着退出程序。如果测试成功，args 会从左右两端同时减少。仅当所有测试都成功并且 args 变得少于两个元素（程序会把包含 0 或 1 个元素的数组当作是回文的）时，程序输出"palindrome"，然后退出。尽管这个程序进行了不少数组处理，但它并没有分配任何内存——它只是预分配了数组 args，然后对它进行收缩。

4.1.9　扩张

通过使用追加操作符~=，可以实现一个数组的扩张。例如：

```
auto a = [87, 40, 10];
a ~= 42;
assert(a == [87, 40, 10, 42]);
a ~= [5, 17];
assert(a == [87, 40, 10, 42, 5, 17]);
```

数组扩展有一些微妙，即数组有可能会进行重新分配。例如，下面的情况：

```
auto a = [87, 40, 10, 2];
auto b = a;              // 现在 a 和 b 引用到相同区块
a ~= [5, 17];            // 追加到 a
a[0] = 15;               // 修改 a[0]
assert(b[0] == 15);      // 是通过还是失败呢?
```

事后对 a[0]的追加赋值也会影响到 b[0]，换句话说，a 和 b 在重新分配之后仍然共享数据，是这样吗？简短的回答为 b[0]可能是 15 也可能不是——D 语言无法做出保证。

现实情况是，没有办法让 a 的结尾处总是拥有足够的空间用于对它进行就地重分配。至少

自在读书 www.zizidiary.com

有的时候，重新分配一定会发生。有一种很容易摆脱此问题的方法，在使用~=对 a 进行追加时总是对它进行重新分配，这样就使得 a ~= b 与 a = a ~ b 总是一致的，即“分配一个新数组，它由 a 与 b 拼接后的内容组成，其后将 a 绑定到这个新数组上。”虽然这种行为容易实现，但它存在很严重的效率问题。例如，数组时常需要在循环里迭代地增长：

```
int[] a;
foreach (i; 0 .. 100) {
    a ~= i;
}
```

对于 100 个元素，任何扩张方法都可以很好地工作，但是当数组变得更大时，只有可靠的解决方案才能保持合理的快速。有一种不太讨人喜欢的方法就是支持方便但不高效的扩张语法 a ~= b。我们鼓励将它用于短数组，但不鼓励因其他语法不太方便而将其用于大型数组。最好的情况是，有一个最简单、最直观的语法对于长短数组都适合。

D 语言给了~=充分的自由，即可以通过重新分配进行扩张，也可以在当前数组末尾存在足够的未使用的内存时，不失时机地进行就地扩张。~=的具体实现完全决定了这种行为是什么样子，但可以保证客户代码能够获得基于同一数组的大量追加所产生的一个平均性能。

图 4-4 演示了扩张请求 a ~= [5, 17]可能出现的两种输出。

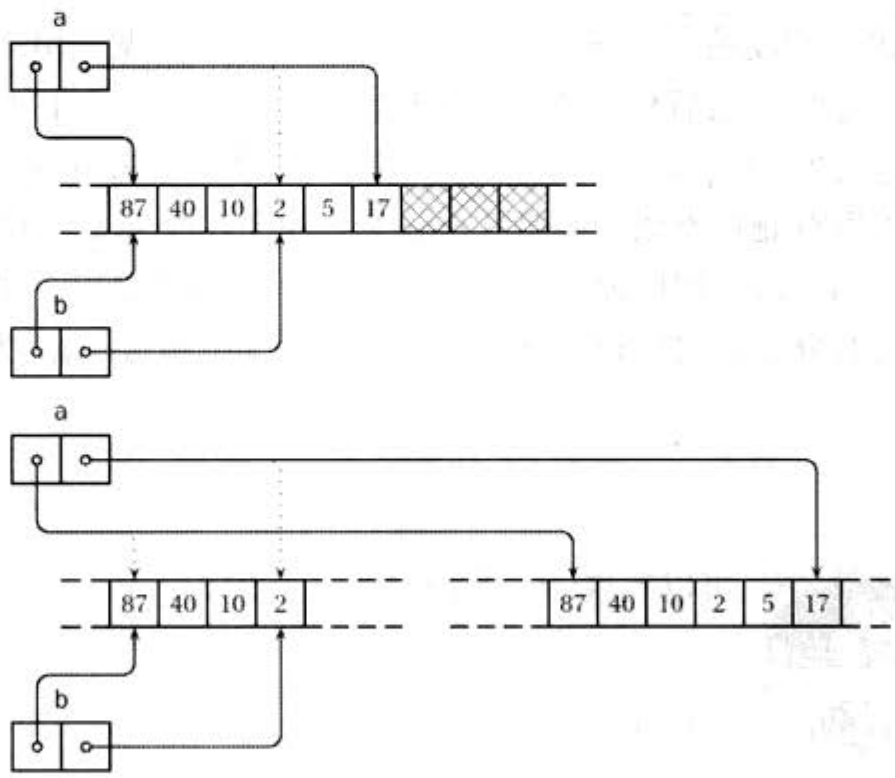


图 4-4　扩展数组 a 可能出现两种输出。第一种情况（上）：在内存块的末尾有足够的内存空间，此时进行就地扩张。第二种情况（下）：没有足够的空间，因此会分配新的内存块，并将 a 调整来引用到它。因此，在扩张之后，a 和 b 的内存块可能会出现交叉也可能不会

一个数组可以有多种扩张方式，具体依赖于底层内存分配器的工作原理。

- 通常，分配器只会分配特定大小（如 2 的幂）的内存块，因此有可能出现这样的情况：请求的是 700 字节，但可能会接收到 1024 字节的空间，其中便会多出 324 字节。当有扩张请求发生时，该数组就会检查是否有多余的存储空间，并使用它。
- 如果没有多余空间，则数组会与底层内存分配器进行协商，初始化一个更大的空间。“你看，

自在读书 www.zizidiary.com

我就固定在这里，在我的右边还有一些空间可以使用。可提供给连接块使用吗？”分配器可能会在当前块的右边找到一个空的内存块，并将它合并成自己的一部分。此操作名叫聚结（coalescing）。这样扩张在没有移动任何数据的情况下仍然可以进行。

- 最后，如果在当前块里没有足够的空间，则数组会分配一个崭新的内存块，然后将它的所有数据复制进去。而编译器实现这时（如它检测到同一个数组会进行重复的扩张）又会故意多分配一些空余的空间。

扩张数组绝不允许重写已有的数组。例如：

```
int[] a = [0, 10, 20, 30, 40, 50, 60, 70];
auto b = a[4 .. $];
a = a[0 .. 4];
// 在这里 a 和 b 是连接在一起的
a ~= [0, 0, 0, 0];
assert(b == [40, 50, 60, 70]); // 通过，a 被重新分配
```

上面的代码是精心调整过的，好让 a 认为在其末尾还有空间：开始时，a 会接收到一个更大一些的空间，接着 b 接收到的是 a 的后半部分，而 a 则减小到它的前半部分。在往 a 追加内容之前，原数组占用的是 a 与 b 的连接块。追加之后的那个 assert，能够确认 a 实际上已重新进行了分配，而不是就地扩张的。只有在能够证明正在进行扩张的那个数组其右边没有其他数组的时候，追加操作符才会进行就地追加；而不管什么时候，只要存在一丝怀疑，追加操作符都会毫不保守地进行重新分配。

4.1.10　对 .length 赋值

对 array.length 进行赋值让你可以对 array 进行收缩和扩张，具体的操作依赖于新长度与旧长度之间的关系。例如：

```
int[] array;
assert(array.length == 0);
array.length = 1000;                    // 增长
assert(array.length == 1000);
array.length = 500;                     // 收缩
assert(array.length == 500);
```

如果该数组因对 .length 的赋值而增长，则增加的元素会被初始化成 T.init。增长的策略与保证与追加操作符~=（参见 4.1.9 节）一致。

如果该数组因对 .length 的赋值而收缩，则 D 语言保证数组不会被重新分配。事实上，如果 n <= a.length，那么 a.length = n 就等同于 a = a[0 .. n]。（不过此保证并没有暗示以后对该数组的扩张不会出现重新分配。）

也可以使用 .length 进行“读取-改写-写入”的操作。例如：

```
auto array = new int[10];
array.length += 1000;                   // 增长
assert(array.length == 1010);
array.length /= 10;                     // 收缩
assert(array.length == 101);
```

这里没有太多魔法。编译器所做的都是将 array.length ‹op›= b 重写成 array.length = array.length ‹op› b。不过还是有那么一点点小魔法（真的只是一个小戏法）：在被重写过的表

自在读书 www.zizidiary.com

达式里，array 只会被计算一次（这个也只会在 array 是一个真正复杂的表达式时才会密切相关）。

4.2 固定大小的数组

D 语言支持在编译期间已知（声明）大小的数组。例如：

```
int[128] someInts;
```

对于每一个类型 T 和大小 n，类型 T[n]与其他数组是完全不同的。例如 uint[10]与 uint[11]不同，与 int[10]也是不同的。

所有固定大小的数组的值都是在声明的地方静态分配的。如果数组的值是在全局进行定义的，则它可以进入程序每一个线程的数据段里。如果是在函数里进行分配的，则在调用该函数的时候，数组会在它的栈上进行分配。（这就意味着，在函数里定义很大的数组是相当危险的。）不过，如果在函数内部定义数组时带上 static，则该数组是在每个线程的数据段里进行分配的，这样就不会有栈溢出的风险了。

在创建的时候，固定大小的数组 T[n]的值都会被初始化成 T.init。例如：

```
int[3] a;
assert(a == [0, 0, 0]);
```

可以使用字面量来初始化 T[n]：

```
int[3] a = [1, 2, 3];
assert(a == [1, 2, 3]);
```

不过要小心：如果将上面的 int[3]替换成 auto，则 a 的类型将会被推断为 int[]，而非 int[3]。虽然逻辑上[1, 2, 3]的类型看起来应该是 int[3]，而且从某种角度它还比 int[]更“准确”，但事实是动态大小的数组比固定大小的数组更常用到。因此仍将它坚持为固定大小的数组字面量，则可能会防碍它的可用性以及使它成为错愕的源泉。事实上，字面量的使用可能已妨碍了 auto 的正确使用。因此，数组字面量默认为 T[]。如果要求为特定类型，而且 n 与字面量里值的个数匹配的话，则可以为 T[n]。

如果使用一个类型为 T 的单值来初始化类型为 T[n]的固定大小的数组，则整个数组都会被该值填充：

```
int[4] a = -1;
assert(a == [-1, -1, -1, -1]);
```

如果不想初始化数组，而只是想在运行时填充它，则可以将其初始值指定为 void：

```
int[1024] a = void;
```

这种未初始化的数组对于当做临时缓冲区的大型数组特别有用。不过请小心——未初始化的整数不会引起太大害处，但是间接类型（如多维数组）的值若未初始化则是不安全的。

与动态数组的元素访问方式一样，使用索引操作符 a[i]，就可以访问固定大小数组的元素。其迭代操作实际上与动态数组的迭代也是一样的。例如，创建一个拥有 1024 个随机数的数组就可以像下面这样做：

```
import std.random;

void main() {
```

自在读书 www.zizidiary.com

```
    double[1024] array;
    foreach (i;  0 .. array.length) {
        array[i] = uniform(0.0, 1.0);
    }
    ...
}
```

此循环也可以不通过索引，而是利用 ref 值来使用数组元素：

```
    foreach (ref element; array) {
        element = uniform(0.0, 1.0);
    }
```

4.2.1　长度

很显然，对于固定大小的数组来说，其长度是很关键的，因为它恰好是这种类型的一部分。与动态数组的长度不同，它的.length 特性是只读的，并且是一个静态常量。这就意味着，不管什么时候，只要是需要编译时常量（如定义另一个固定大小数组时指定的长度），就可以使用固定大小的数组的 array.length：

```
int[100] quadrupeds;
int[4 * quadrupeds.length] legs; // 正确，legs 长度为 400
```

在数组 a 的索引表达式里，$可以用来表示 a.length，也是一个编译时表达式。

4.2.2　边界检查

固定大小的数组的边界检查有一个很有趣的地方。无论什么时候，只要是在编译时表达式里进行索引，编译器都会在编译期间检查有效性；如果出现了边界外的访问则拒绝编译。例如：

```
int[10] array;
array[15] = 5; // 错误！数组索引 15 处于 array[0 .. 10]的边界外面！
```

如果表达式是一个运行时值，那么编译时边界检查只能尽力去完成。运行时的边界检查与动态数组的边界检查遵从相同的协议（参见 4.1.2 节）。

4.2.3　分割

从类型为 T[n]的数组里提取任何分割，都会产生一个类型为 T[]的数组，同时不产生副本：

```
int[5] array = [40, 30, 20, 10, 0];
auto slice1 = array[2 .. $];          // slice1 的类型为 int[]
assert(slice1 == [20, 10, 0]);
auto slice2 = array[];                // 等同于 array[0 .. $]
assert(slice2 == array);
```

编译时的边界检查会针对其中为编译时常量的边界进行。

如果从数组 T[n]里提取编译时已知界限为 T[a1 .. a2]的分割，并且请求的是类型为 T[a2 - a1]的数组，则编译器会接受这样的请求。（这里的分割操作——使用了 auto——所产生的默认类型仍然为 T[]。）例如：

自在读书 www.zizidiary.com

```
int[10] a;
int[] b = a[1 .. 7];    // 正确
auto c = a[1 .. 7];     // 正确，c 的类型也为 int[]
int[6] d = a[1 .. 7];   // 正确，a[1 .. 7]被复制进了 d
```

4.2.4　复制和隐式转换

与动态数组不一样的是，固定大小的数组拥有值语义。即是说，像数组复制、传递数组给函数以及从函数返回数组这样一些操作都会复制整个数组。例如：

```
int[3] a = [1, 2, 3];
int[3] b = a;
a[1] = 42;
assert(b[1] == 2);              // b 是 a 的一个独立副本
int[3] fun(int[3] x, int[3] y) {
   // x 和 y 都是参数的副本
   x[0] = y[0] = 100;
   return x;
}
auto c = fun(a, b);             // c 的类型为 int[3]
assert(c == [100, 42, 3]);
assert(b == [1, 2, 3]);         // b 未受 fun 的影响
```

对于大型数组来说，通过值来传递整个数组可能没那么高效，但通过值来传递也有它的很多好处。其中一个好处就是：短数组和按值传递在高性能计算中被频繁使用。另一个好处是：按值传递拥有一个简单的矫正方法——不管什么时候需要引用语义，只需使用 ref 或自动将其转换成 T[]（参考下一段）即可。最后一个好处是：值语义让固定大小数组与语言的其他许多方面保持了一致。（在历史上，D 语言的固定大小的数组也拥有过引用语义，事实表明这个也成为了客户代码中的各类曲解和特例的根源。）

类型为 T[n] 的数组可以隐式转换成类型为 T[] 的数组。由此方式获得的动态数组不会再进行重新分配——它只是简单地对原始数组的边界进行了控制。因此，如果原始数组是在栈上分配的，则这个转换可能是不安全的。隐式转换是为了方便将类型为 T[n] 的固定大小的数组传递到期望的 T[] 的函数。不过，如果函数将 T[n] 作为返回类型，其结果则无法自动转换成 T[]。

```
double[3] point = [0, 0, 0];
double[] test = point; // 正确
double[3] fun(double[] x) {
   double[3] result;
   result[] = 2 * x[]; // 按数组进行运算
   return result;
}
auto r = fun(point); // 正确，r 的类型为 double[3]
```

可以通过 .dup 特性（参见 4.1 节）来复制固定大小的数组，但无法获得类型为 T[n] 的对象；通过复制，获得的是一个类型为 T[] 的动态数组，其中包含的是固定大小数组的副本。这种行为明白地告诉你，不需要复制固定大小的数组——想要获得 a 的副本，只需要使用 auto copy = a 即可。使用 .dup，获得的是固定大小的数组的动态副本。

4.2.5　相等性比较

固定大小的数组可以使用 is 和==进行比较，跟动态数组一样（参见 4.1.5 节）。你也可以

自在读书 www.zizidiary.com

透明地将固定大小的数组和动态大小的数组进行比较：

```
int[4] fixed = [1, 2, 3, 4];
auto anotherFixed = fixed;
assert(anotherFixed !is fixed);      // 不相同（值语义）
assert(anotherFixed == fixed);       // 相同数据
auto dynamic = fixed[];              // 获得 fixed 的全部数据
assert(dynamic is fixed);
assert(dynamic == fixed);            // 显然相等
dynamic = dynamic.dup;               // 创建一个副本
assert(dynamic !is fixed);
assert(dynamic == fixed);
```

4.2.6　拼接

固定大小数组的拼接遵从的规则与动态数组的拼接规则（参见 4.1.6 节）很相似。其中有一个差异比较重要。如果请求的是固定大小的数组，则获得的也是固定大小的数组；否则，获得的是最新分配的动态数组。例如：

```
double[2] a;
double[] b = a ~ 0.5;    // 将 double[2]与值进行拼接，并得到 double[]
auto c = a ~ 0.5;        // 同上
double[3] d = a ~ 1.5;   // 正确，显式请求固定大小的数组
double[5] e = a ~ d;     // 正确，显式请求固定大小的数组
```

无论什么时候将固定大小的数组请求为连接操作符“～”的结果，都不会出现动态分配——该结果是静态分配的，并且连接后的结果会被复制到它里面。

4.2.7　数组运算

对静态数组的数组运算类似于动态数组的运算（参见 4.1.7 节）。在可能的情况下，编译器会对与数组表达式里所涉及的静态长度相关的数组进行编译时边界检查。在表达式里可以将固定大小的数组和动态数组混合使用。

4.3　多维数组

因为 T[]是一个动态数组，其元素类型为 T；而 T[]本身也是一个类型，这就很容易推断出 T[][]是一个 T[]类型的数组，或换一种说法，即它是一个 T 类型数组的数组。外层数组的每一个元素都依次是一个提供常规数组基本操作的数组。让我们对 T[][]做一个测试：

```
auto array = new double[][5];  // 5 个 double 数组构成的数组，
                               // 每一个数组都被初始化成 null
// 生成三角矩阵
foreach (i, ref e; array) {
   e = new double[array.length - i];
}
```

上面所定义的数组的开头是一个三角形：第一行有 5 个 double 元素，第二行有 4 个，以

自在读书 www.zizidiary.com

此类推，直到第五行（技术上是行 4），这一行只有一个元素。多维数组可以通过简单地将动态数组进行组合获得，因为它们的行可以假定为任意长度（与之相对的是期望右边为直线，这个可以通过让所有行都拥有相同的长度来获得），所以也被称为“不规则数组”（jagged array）。图 4-5 展示的是数组在内存里的安放情况。

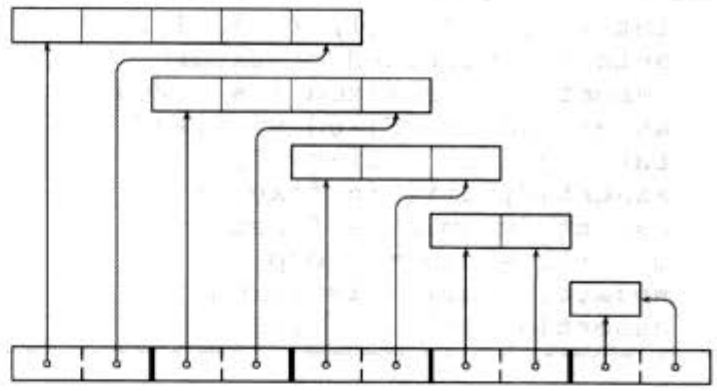


图 4-5　一个不规则数组，它存储的是一个三角矩阵（其定义是在前一页的示例里）

想要访问不规则数组里的元素，可以采用依次在每一维指定索引值的方法，如 `array[3][1]` 访问的是不规则数组里第 4 行里的第 2 个元素。

不规则数组不是连续的。从正面来讲，不规则数组可以在内存里自我伸展，而且需要的连续内存数量更小。这种可以存储不同长度行的能力也能节省大量的内存。从负面来讲，对于多行少列这种“高而瘦”的数组，又会导致大量的开销，因为每行都需要保留一个数组。例如，有一个拥有 1 000 000 行而每一行只有 10 个整数的数组，用于存储一个由 2 000 000 个字[①]（每行一个数组）构成的数组，以及外加的 1 000 000 个小内存块的管理开销；根据内存分配器具体实现的不同，还有可能出现每行的有效负载小于 10 个整数（40 字节）的情况。

不规则数组可能会存在访问效率和缓存不友好的问题。每一次的元素访问都需要两个间接操作：首先是通过与行相对应的外层数组，接着是通过与列相对应的内层数组。如果先获得行然后再使用它，那么按行迭代就不会有什么大问题；但是，如果通过按列迭代来遍历一个不规则数组，则是一件极难办到的事。

如果在编译期间已知列的数目，则可以轻易地将固定大小的数组与动态数组相结合：

```
enum size_t columns = 128;
// 定义一个 64 行 128 列的矩阵
auto matrix = new double[columns][64];
// 不需要对每一行进行分配——它们原本已经存在
foreach (ref row; matrix) {
    ... // 应用类型为 double[columns]的 row
}
```

上面示例中，在 `foreach` 里使用 `ref` 非常关键。没有 `ref` 的话，`double[columns]` 的值语义（参见 4.2.4 节）便会为正被迭代的每一行创建一个副本，这个有可能会拖慢代码执行的速度。

如果在编译期间行和列的数目都已知，则可以像下面这样使用由固定大小的数组所构成的固定大小的数组：

```
enum size_t rows = 64, columns = 128;
// 分配一个 64 行 128 列的矩阵
double[columns][rows] matrix;
// 根本不需要对数组进行分配——它是一个值
foreach (ref row; matrix) {
    ... // 应用类型为 double[columns]的 row
}
```

① 这里指的是数据大小单位，1 个字等于 2 个字节。——译者注

自在读书 www.zizidiary.com

要访问位于 i 行和 j 列上的元素，可以写成 matrix[i][j]。这里有一个比较奇特的地方，数组的声明是以自右向左的顺序指定每一维的大小（即 double[columns] [rows]），但是当访问元素时，索引又是以自左向右的顺序。这是因为[]和[n]都是自右向左按字节进行绑定的，但在表达式里，它们是自左向右绑定的。

通过将固定大小的数组和动态数组进行组合，可以建立各式各样的多维数组。例如，三维数组 int[5][][15]就由 15 个数组构成的，每一个都是动态分配的数组块，而每一块又是由 5 个整数构成。

4.4　关联数组

一个数组可以当做一个函数，它能将正整数（索引值）映射成某些任意类型的值（存储在数组里的数组）。此函数的定义只针对从 0 到数组长度减 1 这个范围的整数，并且函数完完全全地被数组的内容制成了表格的形状。

从这个角度来看，关联数组引入了一种更为一般的数组。不局限于整数，关联数组几乎可以让任意类型成为它的域。在这个域里的每一个值，都可能将某一个值映射到另一个不同类型上——有些像数组的槽（slot）。存储方法和关联算法与数组存在不同，但跟数组一样，关联数组提供了根据指定键来快速存储和获得值的操作方式。

关联数组的类型可表示成 V[K]，这里的 K 指的是键类型，而 V 为被关联的值的类型。举例，下面创建并初始化一个将字符串映射到整数的关联数组：

```
int[string] aa = [ "hello":42, "world":75 ];
```

关联数组字面量（参见 2.2.6 节）是一个用方括号括起来的、以逗号隔开的词汇列表，其中词汇的形式为“键：值”。对于上面的情况，字面量的信息已足够让人明白，显式设定 aa 的类型显得有些多余，因此写成下面这样会让人感觉更舒服些：

```
auto aa = [ "hello":42, "world":75 ];
```

4.4.1　长度

对于关联数组 aa，特性 aa.length 产生的是 aa 里所有键的数量（也是值的数量，假定键与值之间是一一对应的关系）。aa.length 的类型为 size_t。

默认构造的关联数组长度为零，可以与 null 进行相等比较。

```
string[int] aa;
assert(aa == null);
assert(aa.length == 0);
aa = [0:"zero", 1:"not zero"];
assert(aa.length == 2);
```

与数组的同名特性不一样，关联数组的.length 是不可写的。不过可以将关联数组写成 null，从而将它清空。

自在读书 www.zizidiary.com

4.4.2 读写槽

想要往 aa 里写入一对新的“键/值”对，或者想要改写当前为该键存储的值，则只需要像下面那样给 aa[key]进行赋值即可：

```
// 创建一个字符串对字符串的关联数组
auto aa = [ "hello":"salve", "world":"mundi" ];
// 改写值
aa["hello"] = "ciao";
aa["world"] = "mondo";
// 创建新的"键/值"对
aa["cabbage"] = "cavolo";
aa["mozzarella"] = "mozzarella";
```

想要在关联数组里读取指定键对应的值，只需要读取 aa[key]即可。(编译器能够分辨出是读取还是写入，并据此调用稍微有些不同的函数)。现在继续上面的示例：

```
assert(aa["hello"] == "ciao");
```

如果在试着读取某个键对应的值时，在关联数组中并未找到该键，则会抛出一个范围无效异常。大部分情况下，在键不存在时抛出异常过于碍眼而没什么用处，因此关联数组提供了一个带有两个参数的 get 方法，它可以带上默认值进行读取操作。在调用 aa.get(key, defaultValue)时，如果在映射里找到了 key，则返回与之对应的值，而 defaultValue 不会被计算；否则，计算 defaultValue 的值，并返回为 get 的结果。

```
assert(aa["hello"] == "ciao");
// 键"hello"存在，因此忽略第二个参数
assert(aa.get("hello", "salute") == "ciao");
// 键"yo"不存在，于是返回第二个参数
assert(aa.get("yo", "buongiorno") == "buongiorno");
```

如果想要安静地测试键是否存在于关联数组中，则可以使用 in 操作符：

```
assert("hello" in aa);
assert("yowza" !in aa);
// 试图读取 aa["yowza"]时，会抛出异常
```

4.4.3 复制

关联数组是完完全全带有影子复制（shallow copying）的引用：对关联数组进行复制或赋值，只会对同一底层数据创建一个新的别名。例如：

```
auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
auto a2 = a1;                          // a1 和 a2 都指向相同的数据
a1["Bob"] = 100;                       // 对 a1 的更改...
assert(a2["Bob"] == 100);              // ...会同样更改 a2...
a2["Sam"] = 3.5;                       // 反之
assert(a1["Sam"] == 3.5);              // ...亦然
```

4.4.4 相等性比较

操作符 is、==和!=都能正常地工作。对于两个相同类型的关联数组 a 和 b，当且仅当 a

自在读书 www.zizidiary.com

和 b 都引用同一个关联数组时（如一个被另一个的副本初始化），表达式 a is b 的结果才会 true。表达式 a == b 会依次使用==来比较两个数组的“键/值”对。要 a 和 b 相等，则它们必须要拥有相等的键集，并且与每个键相关联的值也要相等。

```
auto a1 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
auto a2 = [ "Jane":10.0, "Jack":20, "Bob":15 ];
assert(a1 !is a2);
assert(a1 == a2);
a2["Bob"] = 18;
assert(a1 != a2);
```

4.4.5　移除元素

想要从映射里移除“键/值”对，可以给关联数组的 remove 方法传递该键。

```
auto aa = [ "hello":1, "goodbye":2 ];
aa.remove("hello");
assert("hello" !in aa);
aa.remove("yowza");          // 无影响："yowza"不在 aa 里
```

remove 方法会返回一个布尔值，如果被删除的键在关联数组中，则该值为 true；否则为 false。

4.4.6　迭代

使用之前的 foreach 语句（参见 3.7.5 节），可以实现关联数组的迭代。“键/值”槽的迭代顺序并不确定：

```
import std.stdio;

void main() {
   auto coffeePrices = [
      "french vanilla" : 8.75,
      "java" : 7.99,
      "french roast" : 7.49
   ];
   foreach (kind, price; coffeePrices) {
      writefln("%s costs $%s per pound", kind, price);
   }
}
```

上面的程序会输出：

```
french vanilla costs $8.75 per pound
java costs $7.99 per pound
french roast costs $7.49 per pound
```

为获得数组里所有键的副本，可以使用 .keys 特性。对于类型为 V[K] 的关联数组 aa，通过 aa.keys 返回的类型为 K[]。

```
auto gammaFunc = [-1.5:2.363, -0.5:-3.545, 0.5:1.772];
double[] keys = gammaFunc.keys;
assert(keys == [ -1.5, 0.5, -0.5 ]);
```

类似地，对于类型为 V[K] 的 aa，特性 aa.values 产生的值是一个类型为 V[] 的数组，它存储在 aa 里。通常情况下，推荐使用 foreach 来进行迭代，而不是使用值或键，这是因为

自在读书 www.zizidiary.com

这些特性会分配一个新数组，对于大型关联数组来说这会占用很大的空间。

有两种方法可以迭代关联数组的所有键和值，同时不会创建新数组：aa.byKey()只会遍历关联数组 aa 的键，而 aa.byValue()会遍历所有的值。例如：

```
auto gammaFunc = [-1.5:2.363, -0.5:-3.545, 0.5:1.772];
// 输出所有键
foreach (k; gammaFunc.byKey()) {
   writeln(k);
}
```

4.4.7　将自定义类型作为键

在内部，关联数组针对键使用了散列和排序，从而确保快速获得给定键所对应的值。如果将自定义类型用作关联数组里的键，则需要定义两个特定的方法：toHash 和 opCmp。由于我们还没有学习如何定义自定义类型和方法，因此推迟到第 6 章再讨论。

4.5　字符串

在 D 语言里字符串受到了特殊的对待。在定义 D 语言的早期有两个决定，后来证明是完全正确的。第一个决定是，D 语言将 Unicode 作为其标准的字符集。Unicode 是当今用于定义和表示文本数据最流行和最广泛的标准。第二个决定是，D 语言选择 UTF-8、UTF-16 和 UTF-32 作为本地编码，同时不特别钟情于某一种，也不会阻止你使用其他的编码。

为理解 D 语言是如何处理文本的，我们需要了解一些关于 Unicode 和 UTF 的知识。想要深入了解，*Unicode Explained* [36]是很有用的参考资料，而《Unicode 协会标准文档》则是终极参考资料，当前的第五版正好对应于 5.1 版本的 Unicode 标准[56]。

4.5.1　代码点

关于 Unicode 有一个很重要的事实，一旦理解了它，许多潜在的问题都会迎刃而解，这个事实就是 Unicode 将抽象字符的概念（即代码点，code point）与表示的概念（即编码，encoding）进行了区分。这是一个很重要但经常被忽略的区分，特别是因为众所周知的 ASCII 标准没有将表示进行区分的概述。古老的 ASCII 是将英文文本里普遍使用的每一个字符，外加一些“控制代码”映射到 0～127（即 7 位）的数字。由于在引入 ASCII 的时候，大部分计算机已经使用 8 位字节作为取址的单位，因此根本就不会存在给 ASCII 文本进行“编码”的问题：在一个 8 位字节里用掉 7 位（这就是编码）。剩下的那个位为各种创新性的用法打开了大门，这也导致了各种相互不兼容的扩展出现了“寒武纪大爆炸式”的爆发。

与之相反，Unicode 首先定义了代码点，简单地讲，就是分配给抽象字符的数字。抽象字符“A”分配的数字是 65，抽象字符€分配的数字是 8364，等等。决定哪一个符号能在 Unicode 映射中获得

自在读书 www.zizidiary.com

位置，以及如何为它们分配一个数字，是 Unicode 协会的一项重要任务。这个真是太好了，因为像我们这些人可以直接使用这个映射，而不用去关心对它进行定义和文档记录的细节。

在 5.1 版本，Unicode 的代码点位于 0 和 1 114 111（上限通常表示为十六进制形式，即 0x10FFFF，或者使用 Unicode 特有的拼写方法进行表示，即 U+10FFFF）。关于 Unicode 存在一个比较普遍误解，即只需要 2 个字节就足以表示任何的 Unicode 字符，这或许是因为有些语言基于早期的 Unicode 标准而将处理标准化成 2 个字节。事实上，通过 2 个字节能表示的 Unicode 字符只有 65 536 的 17 倍那么多。（其真相是，大部分更高位的代码点很少用到或者还未进行分配。）

不管怎样，当讨论代码点的时候，没必要去想关于表示的事。在最上层，代码点就是一个巨大的表格化函数，它将 0 到 1 114 111 之间的整数映射到了抽象字符实体。关于数字范围是如何被分配的还有很多细节，但它不会使我们的最上层描述的正确性有所减弱。如何将 Unicode 代码点表示成字节序列则是编码需要关心的事。

4.5.2　编码

如果 Unicode 只是简单地遵从 ASCII 的光荣传统，则它就只能将上限 0x10FFFF 提升到下一个字节，从而得到每一个代码点的简单“3 字节”表示。但是这种表示存在一个问题。英文或派生于拉丁文书写系统里的大部分文本都只使用了代码点（数字）中很窄的一个范围，这会导致空间浪费。常见拉丁文本的存储会膨胀 3 倍。像亚洲书写系统那样的富字母（richer alphabet）能够充分地利用 3 字节，这一点也没什么问题，因为文本里的总符号会少一些（每一个符号的信息更丰富些）。

为了解决空间浪费的问题，Unicode 采用了多种变长的编码方案。这些方案使用了一个或多个窄代码来表示全部的 Unicode 代码点。窄代码（通常为 8 位或 16 位）即众所周知的“编码单元”（code unit）。每一个代码点可以使用一个或多个代码单元表示。

UTF-8 是第一种被标准化的编码。UTF-8（由 Ken Thompson 在一次新泽西的晚餐[47]上发明出来）是几乎成为可靠、巧妙设计的典范。UTF-8 背后的基本思想是使用 1～6 个字节来对任何给定的字符进行编码，在不同长度的编码之间添加控制位来进行区分。对于开始的 127 个代码点，UTF-8 与 ASCII 是一致的。这使得任何 ASCII 文本也同样是有效的 UTF-8 文本，就这一点而言，它真的是一个很不错的改动。对于 ASCII 范围之外的代码点，UTF-8 使用了变长的编码，如表 4-2 所示。

表 4-2　UTF-8 编码。控制位的选择实现了中途同步、错误恢复、反向迭代

代码点（十六进制）	编码（二进制）
00000000-0000007F	0xxxxxxx
00000080-000007FF	110xxxxx 10xxxxxx
00000800-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000-001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000-03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000-7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

自在读书 www.zizidiary.com

到目前为止，已定义的 Unicode 代码点的范围还停留在 0x10FFFF，因此最后两组序列被预留，以便将来使用。目前只有最多 4 个字节的编码才是有效的。

选择的控制位模式有两条很有意义的性质。

（1）非前导字节与前导字节绝不会相等。

（2）第一个字节可以明确地确定一个编码的长度。

第一条性质非常关键，因为它有两个很重要的应用。一个应用是简单同步——如果在传输流中的某一个位置选取一段 UTF-8 传输，则可以很轻易地找到下一个代码点的起始位置：只需要查找下一个最高有效位为 10 的字节。另一个应用是反向迭代——在一个 UTF-8 字符串里，很容易后退操作，同时也不会引起混乱。反向迭代使得 UTF-8 字符串可用于大量的算法（例如，在一个字符串里查找另一个字符串的最后一次出现，则可以实现得很高效）。第二条性质没那么重要，但它可以简化和加速字符串的处理。

理想情况下，常见的代码点应该拥有小值，而不常见的代码点应该拥有大值。如果满足这个条件，那么 UTF-8 就像是一种很优秀的统计编码器，它以更少的字节实现对更多常见符号的编码。对于派生自拉丁文的语言，当然属于这种情况，大部分代码都使用 1 个字节表示，而不常见的字符使用 2 个字节表示。

UTF-16 也是一种变长编码，但它使用了不同（更优雅）的方法来进行编码。在 0 到 0xFFFF 之间的代码点被编码为一个单独的 16 位编码单元，而 0x10000 到 0x10FFFF 之间的代码点是通过一对编码单元(第一个编码单元的范围是 0xD800 到 0xDBFF，第二个编码单元的范围是 0xDC00 到 0xDFFF)进行表示的。Unicode 为支持这种编码，在 0xD800 到 0xDBFF 这段范围内并没有分配任何有效的字符。这两个范围分别叫做“高位代理区”（high surrogate area）和“低位代理区”（low surrogate area）。

对于 UTF-16，有一种比较普遍的批评意见，即它造成了统计上比较少见同时又是最复杂的状况，而这是最需要仔细审查的状况。大部分（当然不是全部）Unicode 字符，即所谓的“基本多语言面”（Basic Multilingual Plane），适用于一个 UTF-16 编码单元，因此大多数 UTF-16 编码都默许一个字符对应于一个编码单元，而对“代理对[①]”则未做有效的测试。更让人迷惑的是，有些语言在开始时让字符串集中支持 UCS-2（UTF-16 的前身，每一个代码点恰好为 16 位），到后来又增加了对 UTF-16 的支持，从而巧妙地废弃了依赖于字符与编码之间一对一映射的代码。

最后，UTF-32 使用的是每一个编码单元占用 32 位的方法，即真正地支持代码点与编码单元之间一对一的映射。也就是说，UTF-32 是最简单也是最容易使用的表示，但它也是最消耗空间的。常见的推荐做法是使用 UTF-8 进行存储，而在处理期间，则根据需要再临时使用 UTF-32。

4.5.3　字符类型

D 语言定义了 3 种字符类型，即 char、wchar 和 dchar，分别用于表示 UTF-8、UTF-16

① 代理对（surrogate pair）：使用两个 16 位编码单元组成的序列来表示单个抽象字符。其中，代理对的第一部分为高代理编码单元，第二部分为低代理编码单元。——译者注

自在读书 www.zizidiary.com

和 UTF-32。它们的.init 值被故意设置成了无效编码：char.init 为 0xFF，wchar.init 为 0xFFFF，而 dchar.init 为 0x0000FFFF。表 4-2 中说明了 0xFF 不可能是任何有效 UTF-8 编码的一部分，而且 Unicode 也特意没有为 0xFFFF 分配任何有效的代码点。

char、wchar 和 dchar 这 3 种字符类型在单独使用时，多数时候它们的行为都类似于无符号整数，有时还可用于存储无效的 UTF 代码点（编译器不是随时都会要求编码有效），但其原意是用做 UTF 代码点。对于普通的 8 位、16 位和 32 位无符号整数，或者对于使用非 UTF 的其他编码的情形，最好是分别使用 ubyte、ushort 和 uint。例如，如果想要使用那些每个 Unicode 字符占用 8 位的代码页，则你的基础构件（building block）需要使用 ubyte，而不是 char。

4.5.4　字符数组+优势=字符串

当字符类型数据汇集成一个数组（像 char[]、wchar[]或 dchar[]）时，编译器和运行支持库能够“明白”正在处理的内容是 UTF 编码的 Unicode 字符串。因此字符数组能够获得普通数组的强大功能和通用性，外加一些支持 Unicode 所带来的好处。

实际上，D 语言已经定义了 3 种字符串类型来对应 3 种字符宽度，这 3 种类型是 string、wstring 和 dstring。它们根本就不是什么特别的类型。事实上，它们是字符数组类型的别名，只是有点变化：字符类型带有修饰符 immutable，从而禁止随意更改字符串里的单个字符。例如，类型 string 是冗长的 immutable(char)[]类型的代名词。在第 8 章的时候我们会讨论 immutable 这样的类型修饰符，不过对于所有宽度的字符串，immutable 的作用都很简单：string，即 immutable(char)[]，就像 char[]一样（wstring 与 wchar[]一样等），不同之处在于，不能给字符串里的单个字符赋新的值：

```
string a = "hello";
char h = a[0]; // 正确
a[0] = 'H';    // 错误！不能对 immutable(char)进行赋值
```

想要更改字符串里的单个字符，则需要通过拼接创建另一个字符串：

```
string a = "hello";
a = 'H' ~ a[1 .. $];  // Fine, makes a == "Hello"
```

为何会有这样的决定呢？毕竟，在上面的情况里，采用分配一个全新的字符串的方法（回想一下 4.1.6 节，“~”总是会分配一个新的数组），而不是去修改一个已有的串，这是很浪费的。不过禁止修改字符串里的单个字符也有几个很好的理由。其中一个原因就是 immutable 简化了 string、wstring 和 dstring 对象被复制和更改时的状态。事实上，immutable 确保了字符串之间不会出现过度的混淆。例如：

```
string a = "hello";
string b = a;           // b 也为"hello"
string c = b[0 .. 4];   // c 为"hell"
// 如果允许这个，则会更改 a、b 和 c 的值
// a[0] = 'H';
// 下面的连接不会更改 b 和 c 的值
a = 'H' ~ a[1 .. $];
assert(a == "Hello" && b == "hello" && c == "hell");
```

自在读书 www.zizidiary.com

对于 immutable 字符，可以有多个变量引用到同一个字符串，同时不用担心对其中一个量的修改会影响到其他量。字符串对象的复制成本很低，这是因为它不需要进行任何特别的复制管理（如热复制或写时复制）。

关于在编码单元一级禁止更改字符串有一个同样强烈的理由，那就是这种更改完全没什么意义。字符串的元素是变长的，大部分时间想要更换的是逻辑字符（代码点），而不是物理上的 char 类型字符（编码单元），因此很少会去对单个的 char 类型字符“动手术”。如果不对单个的 char 类型字符进行赋值，而是集中于处理整个字符串或其中的片段，则编写出正确 UTF 代码会容易很多。D 语言标准库的基调就是要满足将字符串当做一个完整实体来进行处理的要求，而不是集中于索引值和单个字符。不过，UTF 代码没那么容易写入，例如，上面的连接'H' ~ a[1 .. $]在一般情况下就有一个错漏，因为它假定在 a 里的第一个代码点恰好只有一个字节。正确的方式应该是：

```
a = 'H' ~ a[stride(a, 0) .. $];
```

函数 stride（可以在标准库的 std.utf 模块里找到）返回的是一个字符串里从指定位置开始的编码长度。想要使用 stride 和相关的库函数，可以在程序的顶部插入下面一行：

```
import std.utf;
```

在这里，调用 stride(a, 0)会返回 a 中第一个字符（即代码点）的编码长度，我们将传递它来选择可标识第二个字符起始位置的偏移量。

D 语言支持 Unicode 的最直观方式可以在字符串字面量里找到，关于字符串字面量已经在前面了解过了（参见 2.2.5 节）。D 语言字符串字面量能够理解 Unicode 代码点，并根据选择的编码方案（不管是哪一种）自动地对它们进行编码。例如：

```
import std.stdio;

void main() {
   string a = "No matter how you put it, a \u03bb costs \u20AC20.";
   wstring b = "No matter how you put it, a \u03bb costs \u20AC20.";
   dstring c = "No matter how you put it, a \u03bb costs \u20AC20.";
   writeln(a, '\n', b, '\n', c);
}
```

尽管 a、b 和 c 的内部表示有很大不同，但你也不用为此担心，因为你是通过使用代码点以一种抽象的方式表达字面量的。编译器会负责所有的编码细节，因此在最后此程序会输出 3 行一模一样的文本：

```
No matter how you put it, a λ costs €20.
```

字面量的编码是通过它所处的环境来测定的。在上面的情形里，尽管是完全使用相同的方式拼写的，编译器也未进行任何转换成 UTF-8、UTF-16 和 UTF-32（分别对应于类型 string、wstring 和 dstring）的运行时处理，但它还是获得了字面量的变体。如果请求的字面量编码存在歧义，则可以将 c、w 或 d 当中的任何一个作为后缀添加到字面量的后面（如"like that"d），从而分别将该字符串强制编码为 UTF-8、UTF-16 和 UTF-32（参见 2.2.5.2 节）。

foreach 字符串

如果像下面这样迭代任意宽度的字符串：

自在读书 www.zizidiary.com

```
foreach (c; str) {
    ... // 使用 c
}
```

那么 c 将会迭代 str 的每一个编码单元。例如，如果 str 是一个 char 数组（可能带有 immutable），那么 c 的类型为 char。这个是 foreach 数组的一般行为，正是所期望的，但有时对于字符串则不希望这样。例如，输出字符串里的每一个字符，并将它使用方括号括起来：

```
void main() {
    string str = "Hall\u00E5, V\u00E4rld!";
    foreach (c; str) {
        write('[', c, ']');
    }
    writeln();
}
```

上面的程序会产生难看的输出：

```
[H][a][l][l][?][?][,][ ][V][?][?][r][l][d][!]
```

反白显示的“?”（因系统和字体的不同可能存在差异）是控制台遇到一个无效的 UTF 编码时的默认显示。当然，试图单独输出一个 char 类型字符，而它只有与其他 char 类型字符组合起来才有意义，那么这个操作注定是要失败的。

当为 c 指定一个不同类型时，事情就变得很有意思了。例如，将 c 指定为 dchar：

```
//... 同上，只是添加了"dchar" ...
foreach (dchar c; str) {
    write('[', c, ']');
}
```

此时，编译器会根据由 c 的类型所确定出来的表示，自动插入代码，以实现快速转换 str 里每一个编码单元。上面的循环会输出下面的内容：

```
[H][a][l][l][å][,][ ][v][ä][r][l][d][!]
```

这表明双字节字符 å 和 ä 各自都被正确地转换成了一个 dchar，然后被正确输出。如果 c 的类型为 wchar 也可以得到相同的结果，这是因为使用的两个非 ASCII 字符都分别适合于一个 UTF-16 单元，但它不是最通用的情况（代理对可能会被错误地处理）。出于安全的考虑，在关于字符串的循环里使用 dchar 当然是最好的选择。

在上面的情形里，foreach 完成的是从窄表示到宽表示的代码转换，其实两种方式都可以。例如，可以从一个 dstring 类型的字符串开始，每次迭代一个（编码后的）char 类型字符。

4.6　数组的远房表亲：指针

一个数组对象在内存里会通过存储上下边界跟踪一块类型化的对象。指针是“半个”数组——它只跟踪一个对象。因此，指针没有关于一个内存块是否开始和结束的信息。如果从外部获得了这种信息，则可以使用它来四处移动指针，并让它指向邻近的元素。

T 类型对象的指针可以表示成类型 T*，其默认值为 null（即指向无实际对象的指针）。想要将一个指针指向一个对象，可以使用取地址运行符&。想要使用该对象，则可以使用解引用运

自在读书 www.zizidiary.com

行符*（参见 2.3.6.2 节）。例如：

```
int x = 42;
int* p = &x;        // 提取 x 的地址
*p = 10;            // 使用*p 与使用 x 是一样的
++*p;               // 也可以使用常规的操作符
assert(x == 11); // x 通过 p 被更改了
```

指针支持算术运算，这些运算使得它们就像数组内部的游标一样。递增一个指针可以让它指向数组里的下一个元素，递减它则会将它移动到上一个元素。一个指针加上整数 n，则会得到一个指针，指向的是数组里处于当前位置偏移 n 个以后的那个对象。如果 n 为正，则是向右移；如果 n 为负，则是向左移。为简化索引操作，p[n]等同于*(p + n)。最后，获取两个指针差异的运算 p2 - p1 会产生一个整数 n，它满足 p1 + n == p2。

使用 arr.ptr 可以获得数组 arr 的第一个元素的地址。使用 arr.ptr + arr.length - 1 可以获得一个指向非空数组 arr 的最后一个元素的指针，而使用 arr.ptr + arr.length 获得的是一个指针，该指针恰好经过最后一个元素。现举例说明上面的内容：

```
auto arr = [ 5, 10, 20, 30 ];
auto p = arr.ptr;
assert(*p == 5);
++p;
assert(*p == 10);
++*p;
assert(*p == 11);
p += 2;
assert(*p == 30);
assert(p - arr.ptr == 3);
```

可要当心，这种事情很容易失败（除非可以从指针外面访问到数组的边界信息）。所有指针运算都不会进行边界检查——指针的实现仅仅就是长度为一个字（两个字节）的内存地址，而相应的算术运算对你的请求都会视而不见。这样，指针的运算极其快速，但同时也让它变得相当“无知”。指针还不够聪明，它们没办法意识到自己可能正指向另外的独立对象（相比指向数组内部而言）：

```
auto x = 10;
auto y = &x;
++y; // 啊?
```

指针也不知道它们什么时候会越过数组的边界：

```
auto x = [ 10, 20 ];
auto y = x.ptr;
y += 100;               // 啊?
*y = 0xdeadbeef;        // 俄罗斯轮盘
```

写入一个并不指向有效数据的指针本质上等同于跟程序的完整性一起玩俄罗斯轮盘：这些写入可以到达任何地方，并让精心维护的数据甚至代码遭到严重破坏。这个操作让指针具有了内存不安全的特征。

出于这些原因，大家应该尽量避免使用指针，应该更多地选择数组、类引用（第 6 章）、ref 函数参数（参见 5.2.1 节）以及自动内存管理。所有这些都是安全的，能被有效地检查，并且在大部分情况下不会引起显著的效率降低。

自在读书 www.zizidiary.com

事实上，数组是一种非常有用的抽象类型，它们也经过了精心的设计，以瞄准这样一个小目标：成为除指针外最快速的且内存安全的类型。很明显，“光秃的”指针并没有访问到足够多的信息，因而无法解决其自身的问题；但是，数组知道它们的范围信息，因此可以很廉价地证实所有的操作都是在范围之内的。

从上层的角度看，有人可能会认为数组是相当底层的，它们可能旨在实现一个抽象数据类型。与之相反，从下层的角度看，有人可能会认为数组并不是必需的，因为它们可以通过使用指针来实现。对这两种观点的答复就是响亮地说“听我解释”。

数组作为仍然安全的最低层抽象类型还是需要的。如果只提供指针，则 D 语言就无法为构建于指针之上的各种高级的自定义结构提供任何形式的保证。数组也不应成为一项太高层的特性，因为它们是内建的，所以其他所有的内容都可以构建在它们之上。一个好的内建设施应该是底层的和快速的，这样高层的、或许算不上快速的抽象概念才能构建在它之上。概念的抽象绝不会脱离这种方式。

D 语言提供了恰当的子集，名叫 SafeD（参见 11 章），而且编译器也提供了一个开关用于强制使用该子集。正常情况下，大部分的指针操作在 SafeD 里都是不允许的。内建数组就是一个重要的 SafeD 程序（它功能强大且富有表现力）使能器。

4.7　小结和快速参考

表 4-3 总结的是动态数组的运算，表 4-4 总结的是关于固定大小的数组的运算，表 4-5 总结的是为关联数组提供的运算。

表 4-3　动态数组的操作（a 和 b 是类型为 T[]的两个值，$t, t_1, \cdots, t_k$ 都是类型为 T 的值，而 n 为一个可转换成 size_t 类型的值）

名称	类型	描述
new T[n]	T[]	创建一个数组（参见 4.1 节）
[t_1, t_2, ..., t_k]	T[]	数组字面量，T 被推断为 t_1 的类型（参见 2.2.6 节、4.1 节）
a = b	T[]	将一个数组赋值给另一个数组（参见 4.1.4 节）
a[‹e›]	ref T	通过索引访问一个元素（在‹e›里，$可以用来替换 a.length，其中‹e›必须可转换到 size_t，并且有‹e› < a.length）（参见 4.1 节）
a[‹e_1› .. ‹e_2›]	T[]	从 a 里提取一个分割（在‹e_1›和‹e_2›里，$可以用来替换 a.length，其中‹$e_1$›和‹$e_2$›必须可转换到 size_t，并且有‹$e_1$› <= ‹$e_2$› && ‹$e_2$› <= a.length）（参见 4.1.3 节）
a[]	T[]	参与数组表达式（参见 4.1.7 节），否则，恰好等同于 a[0 .. $]
a.dup	T[]	复制数组（参见 4.1 节）
a.length	size_t	读取数组的长度（参见 4.1.10 节）
a.length = n	size_t	更改数组的长度（参见 4.1.1 节）

自在读书 www.zizidiary.com

续表

名称	类型	描述
a is b	bool	比较数组的同一性（参见 4.1.5 节、2.3.4.3 节）
a !is b	bool	等同于!(a is b)
a == b	bool	逐个元素地比较数组的相等性（参见 4.1.5 节、2.3.12 节）
a != b	bool	等同于!(a == b)
a ~ t	T[]	将数组与一个值进行拼接（参见 4.1.6 节）
t ~ a	T[]	将一个值与数组进行拼接（参见 4.1.6 节）
a ~ b	T[]	拼接两个数组（参见 4.1.6 节）
a ~= t	T[]	往数组里追加一个元素（参见 4.1.6 节）
a ~= b	T[]	往数组里追加另一个数组（参见 4.1.6 节）
a.ptr	T*	得到 a 的第一个元素的地址（不安全）（参见 4.6 节）

表 4-4 固定大小数组的操作（a 和 b 是类型为 T[] 的两个值，t, t_1, …, t_k 都是类型为 T 的值，而 n 为一个可转换成 size_t 类型的静态已知的值）

名称	类型	描述
[t_1, ..., t_k]	T[k]	数组字面量，但类型 T[k] 是显式要求的，T 被推断为 t_1 的类型（参见 2.2.6 节、4.1 节）
a = b	ref T[n]	完整的内容复制（参见 4.2.4 节）
a[‹e›]	ref T	通过索引访问一个元素（在‹e›里，$可以用来替换 a.length，其中‹e› 必须可转换到 size_t，并且有‹e› < a.length）（参见 4.1 节）
a[‹e_1› .. ‹e_2›]	T[]/T[k]	从 a 里提取一个分割（在‹e_1›和‹e_2›里，$可以用来替换 a.length，其中‹$e_1$›和‹$e_2$›必须可转换到 size_t，并且有‹$e_1$› <= ‹$e_2$› && ‹$e_2$› <= a.length）（参见 4.2.3 节）
a[]	T[]	参与数组表达式（参见 4.1.7 节），或者将 a 转换一个动态数组，等同于 a[0 .. $]
a.dup	T[]	复制数组（参见 4.2.4 节）
a.length	size_t	读取数组的长度（参见 4.2.1 节）
a is b	bool	比较数组的同一性（参见 4.2.5 节、2.3.4.3 节）
a !is b	bool	等同于!(a is b)
a == b	bool	逐个元素地比较数组的相等性（参见 4.2.5 节、2.3.12 节）
a != b	bool	等同于!(a == b)
a ~ t	T[]	将数组与一个值进行拼接（参见 4.2.6 节）
t ~ a	T[]	将一个值与数组进行拼接（参见 4.2.6 节）
a ~ b	T[]	拼接两个数组（参见 4.2.6 节）
a.ptr	T*	得到 a 的第一个元素的地址（不安全）

自在读书 www.zizidiary.com

表 4-5　关联数组的运算（a 和 b 两个都是类型为 V[K]的值，k, k_1, …, t_i 都是类型为 K 的值，而 v, v_1, …, v_i 都是类型为 V 的值）

名称	类型	描述
[t_1:v_1, ..., t_i:v_i]	V[K]	关联数组字面量；K 被推断为 k_1 的类型，而 V 被推断为 v_1 的类型（参见 2.2.6 节、4.4 节）
a = b	V[K]	将 b 赋值给 a（参见 4.4.3 节）
a[k]	V	通过索引访问元素（如果 k 未找到，则抛出一个异常）（参见 4.4.2 节）
a[k] = v	V	将值 v 与键 k 相关联（会改写上一个关联，如果存在的话）（参见 4.4.2 节）
k in a	V*	在 a 里查找 k，如果 a 不存在，则返回 null；如果 k 存在，则返回一个指针该值的指针（参见 4.4.2 节）
k !in a	bool	等同于!(k in a)
a.length	size_t	读取 a 里元素的数目（参见 4.4.1 节）
a is b	bool	比较两个关联数组的同一性（参见 4.4.4 节、2.3.4.3 节）
a !is b	bool	等同于!(a is b)
a == b	bool	逐个元素地比较数组的相等性（参见 4.4.4 节、2.3.12 节）
a != b	bool	等同于!(a == b)
a.remove(k)	bool	移除 k 的关联，如果存在的话；仅当 k 在 a 中存在时才返回 true（参见 4.4.5 节）
a.get(k, v)	V	返回 a 里与键 k 相对应的值，v 为默认值（参见 4.4.2 节）

自在读书 www.zizidiary.com

第 5 章

数据与函数、函数式

就连对象已不算是什么新闻的今天，还要谈论数据和函数——仿佛又回到了 20 世纪 70 年代。如果某一天我们只需要将所想要做的内容告诉计算机，然后由它找出具体的实现方法，那是再好不过了。然而，这种想法仍然遥不可及。在实现这个梦想之前，函数仍然是所有主要编程准则中不可或缺的组成部分。在本质上，任何程序都是由操作数据的计算构成的。所有我们精心建造的工具（如类型、对象、模块、框架、设计模式）都仅仅是为计算增加一些有意义的特性（如模块化、错误隔离或者易维护性）。一门优秀的语言应允许其用户在用于“做什么”的代码与用于“是什么”的代码之间获得一个黄金比例。理想的比率取决于多种因素，而程序的规模很显然便是其中一个：简短的脚本应主要关注于完成具体的事情，而大型的应用程序理所当然关心特定的非可执行性方面的事情，如接口、协议和模块化约束。

D 语言支持创建各种规模的应用程序，因此它拥有强大的模块化能力。在合理的情况下，它会尽量减少用于“是什么”的代码，从而让你可以更专注于“做什么”的部分。精心编写的 D 语言函数通常都很紧凑且通用，其压缩率有时甚至让人难以致信。接下来便详细讲解这些内容。

5.1 编写一个简单函数并进行单元测试

也许有人会说，计算机大部分时间做的事就是搜索（一些乏味的事情除外，如等待输入）。数据库程序要进行搜索。人工智能程序要进行搜索。又是哪个烦人的自动电话银行职员在跟你闲聊呢？搜索一下吧。还有互联网搜索引擎……没错，这些你都知道。正如你通过经验直接了解到的确信无疑的信息：许多程序表面上与搜索无任何关系，但在其核心却执行了相当多的搜索操作。不管怎样，这里还要解决一个问题：在这个解决方案中涉及了搜索。许多精致的问题解决方案都智能地依赖搜索，以及依赖于通过设置更容易进行的搜索。毫无疑问，计算世界充满了与搜索相关的基础要素：模式匹配（pattern matching）、关系代数（relational algebra）、折半查找（binary searching）、散列表（hashtable）、二叉树（binary tree）、字典树（trie）、红黑树（red-black tree）、

自在读书 www.zizidiary.com

跳跃列表（skip list）……没错，在这里我们不会假装要查看全部内容，只需要瞄准一个合适的目标即可——先从最简单的线性搜索开始，使用 D 语言来定义几个简单的搜索函数。事不宜迟，现在就来编写一个函数，让它能辨别出在一组整数里是否包含某个特定的整数。

```
bool find(int[] haystack, int needle) {
    foreach (v; haystack) {
        if (v == needle) return true;
    }
    return false;
}
```

完全没问题。既然这是我们遇到的第一个函数，那么这里就详尽地描述一下它实际完成的工作。当编译器看到 find 的定义时，它会将该函数编译成二进制代码。在运行过程中，当调用 find 时，haystack 和 needle 会以值的方式传入函数。在这里，“传值”（by value）并不是说你在传递一个拥有上百万个元素的数组时也会进行复制。如第 4 章所讨论的，类型 int[]（即整数数组的一个分割）被称作“胖指针”（fat pointer），它实际上是一个“指针+长度”对或“指针+指针”对——它存储的内容仅限于一个数组的特定部分。正如 4.1.4 节所解释的，往 find 里传递一个拥有上百万个元素的数组分割，就意味着需要传递足够的信息以便获得此数组分割的起始和结束位置。通过一个小的、带边界的表示（它知道如何在容器里游走）来处理容器的方法在 D 语言及其标准库里随处可见。这种表示常被称为“范围”（range）。因此，最后也就只有 3 个机器字需要从调用者传递进 find。一旦 find 获得控制权，它就会执行其动作，然后返回一个布尔值（通常是在一个处理器寄存器里），在这时调用者也正准备好提取它。正如在家居装饰展示中完成一项令人难以置信的复杂任务之后，他们常会在表演的结尾鼓舞人心地说：就这么回事！

坦率地讲，find 的设计还不够完美。返回布尔值并不能提供足够丰富的信息。通常，还需要位置信息，例如，想继续搜索。这时，可以返回一个整数（像-1 那样的特殊值可用于表示“未找到”）。不过，尽管整数非常适合于指代连续存储区域里的元素，但它们对于其他容器（如链表）却是非常低效的。通过 find 获得链表的第 n 个元素需要从该链表的表头一个节点接一个节点地进行遍历——跟 find 本身的操作差不多！因此，对除数组以外的大部分类型来说，返回一个整数都不太适合。

有一种想法可能适用于各种类型的 haystack，如数组、链表，甚至还有文件和套接字（socket）。即让 find 只是简单地从 haystack 里挨个取出元素，一直到找到要搜索的那个值为止，然后返回 haystack 剩下的内容。如果 find 未找到该值，它自然会返回一个空的 haystack。它可简单、通用地描述为：find (haystack, needle) 会从左边逐步将 haystack 进行缩减，一直到它起始于 needle，或者一直到 haystack 为空，然后返回 haystack 的剩余部分。我们现在来实现一个用于 int[] 的设计：

```
int[] find(int[] haystack, int needle) {
    while (haystack.length > 0 && haystack[0] != needle) {
        haystack = haystack[1 .. $];
    }
    return haystack;
}
```

请注意，find 只访问了 haystack 的第一个元素，并且重复地将 haystack 更短一些的

自在读书 www.zizidiary.com

子集赋给 haystack。这些原语以后应该很容易被替换掉，如替换成列表相关的原语，稍后我们会考虑通用性问题。现在就来试一下 find。

近年来，几乎在所有开发方法中，人们都逐渐开始关注进行适当的软件测试。这是一种不错的趋势，因为适当的测试对错漏管理有着非常积极的影响。既然这样，我们也与时俱进，为 find 编写一个单元测试。将下面的内容简单地放置在 find 之后（如果你完全想按自己的意思来）或之前（如果是一名真正的测试驱动开发人员）即可：

```
unittest {
    int[] a = [];
    assert(find(a, 5) == []);
    a = [ 1, 2, 3 ];
    assert(find(a, 0) == []);
    assert(find(a, 1).length == 3);
    assert(find(a, 2).length == 2);
    assert(a[0 .. $ - find(a, 3).length] == [ 1, 2 ]);
}
```

为获得一个工作模块，我们必须要做的是，将此函数和单元测试放置到一个叫 searching.d 的文件里，然后运行下面的命令：

```
$ rdmd --main -unittest searching.d
```

如果传递了-unittest 标志，则单元测试代码会被编译，并会在主程序之前运行；否则，编译器会忽略掉 unittest 块，如果你在没有启动延迟的情况下运行已测试过的代码，那这个会很有用。标志--main 会指示 rdmd 添加一个什么也不做的 main 函数。如果你忘记了--main，也不用着急。连接器会使用其本地语言——加密后的克林贡语，流畅和详尽地提醒你[①]。那个 main 函数代理对我们非常有用，因为我们只对单元测试感兴趣，对实际的程序则不关心。也许会有大量的开发人员对此感兴趣，他们可以提取这个小文件，并在自己的项目里使用，而每一个项目还需要定义自己 main 函数。

5.2　传递约定和存储类别

正如前面提到的，find 的两个参数（一个是 int，另一个是表示 int[]的胖指针）都会被复制成 find 的私有状态。当 find 返回的时候，有一个胖指针会复制回调用者。这一点很容易被认为是通过传值直接进行调用的。尤其是，对这些参数的更改，它们在函数返回之后都无法被调用者“看到”。不过请注意间接操作（indirection）：由于分割的“内容”不会被复制，因此对分割中单个元素的更改则会被调用者“看到”。例如：

```
void fun(int x) { x += 42; }
void gun(int[] x) { x = [ 1, 2, 3 ]; }
void hun(int[] x) { x[0] = x[1]; }
unittest {
    int x = 10;
    fun(x);
```

① 请参考本书 Scott Meyers 序里的注释。——译者注

自在读书 www.zizidiary.com

```
    assert(x == 10);                    // 未更改
    int[] y = [ 10, 20, 30 ];
    gun(y);
    assert(y == [ 10, 20, 30 ]); // 未更改
    hun(y);
    assert(y == [ 20, 20, 30 ]); // 已更改!
}
```

这是怎么回事呢？在开始的两种情况里，fun 和 gun 只会更新它们自己的参数的副本。特别是在第二种情况里，它将那个胖指针进行了重新绑定，以指向其他某个地方，对原始数组未做任何改动。不过，在第三种情况里，hun 选择了更改数组里的某个元素，这个更改会反映到原始数组中去。当我们认为分割 y 与它所管理的 3 个整数是处于不同的地方时，其中的原因就不难想像了。因此，如果你一次性赋值一个分割，如 x = [1, 2, 3]，那么 x 之前所占用的分割会被丢弃，同时创建一个新的分割。但如果你更改了分割中的某个特定元素 x[i]，那么其他正“盯着”该元素的分割（在这里，指的是 hun 的调用者）将会“看到”这个变更。

5.2.1　ref 参数与返回

有时我们想要让调用者也能看到这种更改，这时可以借助于存储类别 ref：

```
void bump(ref int x) { ++x; }
unittest {
    int x = 1;
    bump(x);
    assert(x == 2);
}
```

如果一个函数期望的是一个 ref，那么它只会接受“真正意义上的”数据，而非临时变量。在编译期间，所有非左值内容都会被拒绝。例如：

```
bump(5); // 出错！不能将一个右值绑定到 ref 参数
```

有时人们会认为工作已完成，而实际上这个调用并没有明显的效果，这时就会犯下严重的错误。你也可以将 ref 附着到函数的结果上。在这种情形下，函数的结果自身就是一个左值。例如，我们将 bump 调整为下面的样子：

```
ref int bump(ref int x) { return ++x; }
unittest {
    int x = 1;
    bump(bump(x)); // 两将递增
    assert(x == 3);
}
```

内层对 bump 的调用会返回一个左值，该值然后又成为外层调用的一个合法参数。如果像下面那样定义 bump，那么编译器将拒绝调用 bump(bump(x))，因为试图将 bump(x) 产生的右值绑定到外层 bump 调用的 ref 参数上是非法的：

```
int bump(ref int x) { return ++x; }
```

自在读书 www.zizidiary.com

5.2.2　in 参数

如果对某个参数指定了 in，那么这个参数就会被认为是只读的，因此也就不能以任何方式更改它。例如：

```
void fun(in int x) {
   x = 42; // 出错! 不能修改'in'参数
}
```

上面的代码无法编译，这表明 in 是相当严格的。尽管函数 fun 已拥有了该参数的一个副本，但它还是不能更改这个参数。

在检查函数的实现时，如果在其内部拥有一个实际上完全恒定的参数，肯定是很有用的，但更有意义的影响还是要属这个函数的外面。in 参数不允许对这个参数进行更改，即使是间接的也不行；也不允许在函数返回之后会有更改反映到该对象内部。这让 in 参数非常有用，因为它们不仅仅只对函数实现自身提供了保证，同时也向其调用者提供了保证。例如：

```
void fun(in int[] data) {
   data = new int[10]; // 出错! 不能修改'in'参数
   data[5] = 42;       // 出错! 不能修改'in'参数
}
```

第一个错误并不奇怪，因为它与单独涉及 int 的错误一样。第二个错误则更有意思。编译器居然神奇地将 in 从 data 本身扩大了 data 的每个槽——in 具有“深度延展性”。

实际上这种限制并不仅限于一级，可以延展到任何深度。现在就用一个多维数组来举例说明：

```
// 一个 int 类型的数组的数组拥有双重间接性
void fun(in int[][] data) {
   data[5] = data[0];       // 出错! 不能修改'in'参数
   data[5][0] = data[0][5]; // 出错! 不能修改'in'参数
}
```

因此，in 会保护其数据不被连带修改，哪怕是各种间接性的修改也不行。这种行为并不特定于数组，它也适用于 D 语言的所有数据类型。事实上，在参数的上下文里的 in 就是 const 类型修饰符的同义词。在第 8 章会详细讨论 const 是如何发挥作用的。

5.2.3　out 参数

有时 ref 参数仅仅是表示该函数会将某些内容存入进去。在这种情形下，你也可以使用存储类别 out。它与 ref 很像，不同之处在于它会在进入函数时将这个参数初始化成默认值：

```
// 计算 a 除以 b 的商和余数
// 通过值的方式返回商值，而余数保存在'rem'参数里
int divrem(int a, int b, out int rem) {
   assert(b != 0);
   rem = a % b;
   return a / b;
}
unittest {
```

自在读书 www.zizidiary.com

```
    int r;
    int d = divrem(5, 2, r);
    assert(d == 2 && r == 1);
}
```

上面的代码完全也可以使用 ref 来代替 out。不过，使用 out 能够更清晰地告知调用者：函数 divrem 并不期望 rem 在进入时包含任何有意义的内容。

5.2.4 static 数据

虽然 static（静态的）与函数参数传递无关，但在这里讨论是很恰当的，因为像 ref 一样，应用于数据的 static 也是一个存储类别，它表示的是与数据如何进行存储有关的细节。

任何变量声明都可以使用 static 来进行修正，它表明此数据在每一个执行线程中都会有一个副本。这一点与 C 语言建立的内存分配传统（在整个应用程序里此数据只有一个副本）有所不同。第 13 章会讨论其中的基本原理以及 D 语言的决定所带来的影响。

静态数据可以被共享，在跨函数调用时，不管是处于函数的里面还是外面，它都会保留其值。将静态数据选择放置在各种不同的作用域里，你只需要考虑其可见性，而存储的事可以不管。在模块一级，static 实际上与 private 的处理方式是一样的。

```
static int zeros; // 实际上等同于private int zeros;

void fun(int x) {
    static int calls;
    ++calls;
    if (!x) ++zeros;
    ...
}
```

静态数据必须使用编译时常量进行初始化。为在函数第一次执行时初始化其中的函数级静态数据，可以使用一个简单的技巧，即使用一个伴随有 bool 的 static 变量：

```
void fun(double x) {
    static double minInput;
    static bool minInputInitialized;
    if (!minInputInitialized) {
        minInput = x;
        minInputInitialized = true;
    } else {
        if (x < minInput) minInput = x;
    }
    ...
}
```

5.3 类型参数

现在回到 5.1 节开始部分中定义的 find 函数，因为它存在很多问题。首先，find 的实用性非常狭窄，因此有必要找出方法让它更为通用。先简单地分析一下。出现在 find 里的 int 是硬编码，单纯而又简单。当它用于查找分割 double[] 里的 double 值或者查找 string[] 里的 string 值时，在代码外形上并没有什么不同。于是，我们需要将 int 变成一个占位符——find 的一个参

自在读书 www.zizidiary.com

数，它描述的是所涉及实体的类型，而非具体的值。为此，我们需要将定义调整一下：

```
T[] find(T)(T[] haystack, T needle) {
    while (haystack.length > 0 && haystack[0] != needle) {
        haystack = haystack[1 .. $];
    }
    return haystack;
}
```

正如预想的那样，在 find 的函数体里并没有做任何改变，改变的仅仅是它的签名。其签名现在有两对括号。第一对括号里列出的是函数的类型参数，第二对括号里是常规的参数列表，在这里可以利用刚才定义好的类型参数。现在我们不仅可以处理 int 类型的分割，也可以处理所有其他类型的分割，可以是内建类型也可以是自定义类型。除此之外，我们之前的那个单元测试也可以继续工作，因为编译器会根据参数类型自动推断 T。优雅之极！不过还没到可以舒服地休息一下的时候，现在我们继续添加一个单元测试，以确认这些不同寻常的声明是否正确：

```
unittest {
    // 测试泛型能力
    double[] d = [ 1.5, 2.4 ];
    assert(find(d, 1.0) == null);
    assert(find(d, 1.5) == d);
    string[] s = [ "one", "two" ];
    assert(find(s, "two") == [ "two" ]);
}
```

现在，当编译器看到改进后的 find 定义，会发生什么呢？与 int[] 的情况相比，编译器将面临更为严峻的挑战，因为 T 现在还是未知的——它可以是任何类型。而且不同类型的存储是不相同的，传递的方式也会不同，而且还会出现不同的“==”定义。应对这一挑战是很重要的，因为类型参数真正开启了代码的各种可能性和多样的可重用性。关于通过类型参数化而生成代码，在当今存在两种主流的思想派别[43]。

- 同构转换（homogeneous translation）：将所有数据都转换成一个公共的格式，它允许只编译一个可用于所有人的 find 版本。
- 异构转换（heterogeneous translation）：使用不同的类型参数（如 int、double、字符串）来调用 find 函数，从而提示编译器生成尽可能多的 find 特定版本。

在同构转换时，语言必须以前置条件的形式为 find 函数提供一个统一的数据访问接口。异构转换也非常有用，它好比让你拥有一个助手，对你所提出的每一种数据格式，，它都会为你编写出一个特定的 find，并且所有的构建都源自同一个模具。很明显，这两种方法都有各自的优点和缺点，在各种语言社区，它们都是经常争论不休的话题。同构转换具有一致性、简单性和生成的代码紧凑的特点。例如，传统的函数式语言喜欢以列表格式放置所有的内容，而大量传统的面向对象语言喜欢将所有内容变成一个对象——由它提供对其特性的统一访问。不过，同构转换则可能包含了过于刚性、缺乏表达能力和效率低下这样一些缺点。相比之下，异构转换则能让生成的代码具有专业化程度高、表达力强以及速度快的特点，而它付出的代价则可能包括生成代码雍肿、语言复杂性增加以及编译模型难以掌控（一个广为传播的反对异构方法的说法是：它们把宏给发杨光大了[喘口气]；而且，曾几何时 C 语言已将宏的名声给搞臭了，因此这个标语实际上隐含着强大和消极两个层面的意思）。

自在读书 www.zizidiary.com

这里有一个细节需要注意，即包含关系：异构转换包括了同构转换。其中的原因很简单："多种格式"自然包括了"一种格式"，而且"多个实现"也包括"一个实现"。因此可以认为（其他所有问题先忽略不计）：异构转换比同构转换更强大。如果你已经在使用异构转换，那么当你希望选择统一的数据格式和统一的函数时，至少原则上没有什么能阻止你这样做。在同构转换里，却没那么容易提供逆向选择。不过，就此得出异构的方法"更好"可能有些过于简单，因为除了表达力外，还有其他的观点也是需要考虑的。

D 语言使用的异构转换具有（请注意，技术术语存在争议）静态作用域符号查找和延迟类型检查的特点。即是说，当 D 语言编译器看到泛型的 `find` 定义时，它会解析和保存函数体，记住函数定义的位置，其后什么也不做，一直到 `find` 被调用。这时，编译器会获取解析后的 `find` 定义，并使用调用者选择用来代替 `T` 的类型尝试着编译它。当该函数使用到这些符号时，编译器就会在定义函数的上下文中查找这些符号。

编译器有可能无法为你指定的特定类型生成 `find`，这时会产生一个错误消息。实际上这一点比较烦人，因为该错误有可能是由于 `find` 里某个 bug 未被检测出来所导致的。事实上，这一点正是下一节主要解决的问题，因为 `find` 里有两个 bug——不是功能性 bug，而是通用性 bug：就目前的情况而言，`find` 过于通用，同时又不够通用。接下来看看其中的禅意。

5.4　签名约束

假设有一个 `double` 类型的数组，我们想要在它里面查找一个整数。它应该可以很顺利就找到，真的是这样吗？

```
double[] a = [ 1.0, 2.5, 2.0, 3.4 ];
a = find(a, 2); // 出错! find(double[], int)未定义
```

这样想你就错了。实际情况是：`find` 首先期望的是 `T[]`，其次才是 `T`。然而，`find` 接收到的是一个 `double[]`和一个 `int`，它们会分别声明 `T = double` 和 `T = int`。如果我们尽力窥视一下，就会看到：在这种情形里，调用者的意图是想获得 `T = double` 以及想利用友好的从 `int` 到 `double` 的隐式转换。不过，在一般情况下，让语言同时进行隐式转换和类型推断是一种很危险的做法，因此 D 语言不会同时执行它们。如果你声明的是 `T[]`和 `T`，那么就不能传递 `double[]`和 `int`。

似乎我们实现的 `find` 缺少了通用性，因为它要求分割的类型必须与被搜索值的类型一致。事实上，我们需要接受可对分割元素使用==进行比较的任何值。

当然，如果一个类型参数可行，那么使用两个类型参数则更没问题：

```
T[] find(T, E)(T[] haystack, E needle) {
    while (haystack.length > 0 && haystack[0] != needle) {
        haystack = haystack[1 .. $];
    }
    return haystack;
}
```

现在测试完全成功通过。但是，从技术上讲，`find` 在撒谎，因为它宣称可以接受任何的 `T` 和 `E`，其中也包括了毫无意义的配对！为说明这是一个问题，请看下面的调用：

自在读书 www.zizidiary.com

```
assert(find([1, 2, 3], "Hello")); // 出错!
   // 针对 int[]和 string, 比较"haystack[0] != needle"是无效的
```

编译器发现了一个问题。不过，它是在位于 find 函数体内部的比较操作里发现的。这对于不知情的用户可能造成混淆，因为该错误到底是处于调用端还是在 find 的实现内部，情况并不清楚。尤其是，编译器所报告出来的文件和行都是直接指向 find 的定义内部。如果该问题处于一个很长串的调用链末尾，那么它会让情况变得更加混乱不堪，因此我们需要修复这个问题。现在，这个问题的根源是什么？如果支持一点双关语，那么 find 便可以让其签名进行函数体无法实现的检查。

在其签名里（即恰好在第一个“{”之前的那部分），find 郑重地说明它会接受一个任何 T 类型的分割和一个任何 E 类型的值。编译器则会欣然承认它，同时将这些无意义的参数派发给 find，推导出 T =int 和 E =string，并在以后调用它。然而，一旦涉及 find 函数体，编译器会尴尬地发现它并不能为比较 haystack[0] != needle 生成有意义的代码，然后报告类似于“find 有点嚼不动了”的错误。在 T 和 E 所有可能的组合中只有很少一部分才能真正地被 find 的函数体接受，也就是说，只有那些可支持相等性比较的组合才被接受。

有一种可能的解决方案是构建某种“退避机制”（back-off mechanism）。另一种解决方案（D 语言所选择的）是允许 find 的实现者可以系统地限制函数的适应性，而正确指定约束的地方是在 find 的签名里，就在 T 和 E 第一次出现的地方。D 语言通过“签名约束”（signature constraint）实现了对这一点的支持：

```
T[] find(T, E)(T[] haystack, E needle)
   if (is(typeof(haystack[0] != needle) == bool))
{
   ...// 实现保持一样
}
```

签名里的 if 子句会明确告知：如果表达式 haystack[0] != needle 的类型为 bool，那么 find 会接受 T[]类型的 haystack 和 E 类型的 needle。这里有几个重要的结论。第一个结论是，if 子句会向写入者、编译器和读取者阐明 find 对其参数有什么样的期望，同时完全不用去检查函数体（多数时候它们都会比我们现在的这个函数更长）。第二个结论是，有了 if 子句，当不兼容的参数传递给 find 时，它现在就可以优雅地拒绝承诺，进而允许其他如函数重载那样的功能可以顺利地工作。第三个结论是，新的定义让编译器能产生更好的错误消息，因为错误已在 find 的调用里变得很明显，而不是在函数体里。

请注意，在 typeof 里的表达式是绝不会在运行时进行计算的。其目的仅仅是要辨别出如果该表达式被编译，它会是什么类型。如果 typeof 里的表达式无法编译，它也不会出现编译错误，而只是单纯地给出信息：它根本就没有类型，而“根本没有类型”并不是 bool 类型。尤其是，你不用担心如果 haystack 的长度为零，haystack[0]会进入测试。相反，你不能在签名约束里放置运行时才会计算的条件，例如，不能指定要求将 find 的许可限制为 needle > 0。

5.5　重载

我们定义了用于一个分割和一个元素的 find。现在着手编写 find 的另一个版本，它能

自在读书 www.zizidiary.com

辨别出一个分割能否在另一个分割里找到。一种常见的方法是使用两个嵌套循环进行蛮力搜索。这种算法不是十分高效——其运行时间与两个分割的长度之积成正比。我们不用担心算法的时间效率会是什么样子，我们先集中精力为新增的函数定义一个好的签名。在上一节，我们几乎学到了所需要的一切。事实上，该实现可以编写成这样：

```
T1[] find(T1, T2)(T1[] longer, T2[] shorter)
   if (is(typeof(longer[0 .. 1] == shorter) : bool))
{
   while (longer.length >= shorter.length) {
      if (longer[0 .. shorter.length] == shorter) break;
      longer = longer[1 .. $];
   }
   return longer;
}
```

啊哈！瞧瞧，这一次我们可没再掉进让函数过于特殊的陷阱。较差的定义可能会是这个样子：

```
// 不！此签署存在严重的局限性！
bool find(T)(T[] longer, T[] shorter) {
    ...
}
```

必须承认，它有点简洁，但限制太多。我们的实现可以分辨出一个 int 分割是否包含了一个 long 分割，或者能分辨出一个 double 分割是否包含了一个 float 分割，同时还不会将数据四处复制。这些选择对于简单的签名是无法轻易办到的。你要么需要四处复制数据，以确保正确的类型是处于合适的地方，要么完全放弃使用函数，换作手工进行搜索。那么什么样的函数才是那种在测试示例里看起来可爱，而在正式使用时却遭遇失败的函数呢？

看看这个实现，请注意（在外层循环里）现在已熟悉的那个减除操作——longer 会从左端逐个减少元素。在块比较 longer[0 .. shorter.length] == shorter 里，内层循环是隐性的，它会将 longer 中的前 shorter.length 个元素与 shorter 进行比较。

D 语言支持函数重载（overloading）——几个函数可以共享相同的名称，只要它们拥有不同数量的参数，或者至少有一个不同类型的参数。在编译过程中，语言规则会决定实际调用哪一个。重载是建立在我们天生的根据上下文消除单词歧义的语言能力基础上；在调用者需要记住的词汇量不会出现相应增长的同时，又能提供丰富的功能，重载对此很有帮助。存在的负面风险是：如果调用消除歧义的规则太过宽松，人们就可能会认为他们正在调用某个函数，而实际上调用的却是另一个函数；或者，如果规则太严，那么人们就需要将他们的代码进行“扭曲”，以便向编译器解释他们想要调用的是哪一个版本。D 语言会尽力让规则保持简单，在这种特定情况下，规则实际上是显而易见的：如果一个函数的签名约束（if 子句）计算结果为假，那么此函数会简单地从重载集里消失——它根本就不会被当做候选项。对于我们的那两个 find 版本，相应的 if 子句对于相同的参数绝不会为真。因此，对 find 任何可能的调用，这两个重载中至少会有一个使自己变得不可见，从而绝不会有模棱两可的问题需要解决。最后，让我们使用一个单元测试来验证这些想法：

```
unittest {
   // 对刚引入的重载进行测试
   double[] d1 = [ 6.0, 1.5, 2.4, 3 ];
```

自在读书 www.zizidiary.com

```
    float[] d2 = [ 1.5, 2.4 ];
    assert(find(d1, d2) == d1[1 .. $];
}
```

这两个版本的 find 可能处于同一个文件，也可能处于不同文件。这两者之间并无冲突，因为它们的 if 子句绝不会同时为真。现在，要进一步讨论一下重载规则。假设我们要大量使用 int[]来工作，因此我们想针对该类型定义一个优化的 find 函数：

```
int[] find(int[] longer, int[] shorter) {
    ...
}
```

如前所述，find 的这个重载没有类型参数。同时，也很明确：在我们之前定义的泛型 find 和为整数特别定义的 find 之间存在竞争。在重载链里，两个函数的相对位置是什么呢？它们中的哪一个能获得下面的调用呢？

```
int[] ints1 = [ 1, 2, 3, 5, 2 ];
int[] ints2 = [ 3, 5 ];
auto test = find(ints1, ints2); // 是正确还是错误？是泛型函数还是特化函数？
```

D 语言应对这个问题的方法非常简单：其选择总是会偏向于更特化的函数。现在，在一般情况下，概念“更特化的”还需要进行一点解释：它表明这里有一种特化顺序，即用于函数的“小于”。的确是这样，这种关系也被称为“函数的偏序”（partial ordering of function）。

5.5.1　偏序函数

虽然这个听起来像是需要很厚的数学功底[①]，但偏序是一个非常简单的概念。在我们的函数里，可以将它认为是大家所熟悉的数值关系“≤”对其他集合的一种推广。给定两个函数 foo_1 和 foo_2，我们想要辨别出 foo_1 在适合调用的程度方面是否比 foo_2 更小（我们将“foo_1 的适合程度要小于 foo_2”表示成 $foo_1 \leqslant foo_2$）。如果我们定义这样一种关系，那么就有了一个抉择标准——用于决定在重载竞争中谁应该获得调用：在调用 foo 函数时，就通过“≤”将各种可能的 foo 函数进行排序，然后选择其中找到的“最大的”那个 foo 函数。值得一提的是，偏序必须具有自反性（$a \leqslant a$）、反对称性（如果有 $a \leqslant b$ 和 $b \leqslant a$，那么就认为 a 和 b 是等效的）和传递性（如果 $a \leqslant b$ 和 $b \leqslant c$，那么 $a \leqslant c$）。

D 语言为函数定义了一个简单的偏序关系：如果 foo_1 可以以 foo_2 的参数类型进行调用，那么就有 $foo_1 \leqslant foo_2$。有可能同时出现 $foo_1 \leqslant foo_2$ 且 $foo_2 \leqslant foo_1$，在这种情况下，我们说它们俩是“等同特化的”（equally specialized）。例如：

```
// 三个等同特化函数：每一个都可以使用另一个的参数类型进行调用
void sqrt(real);
void sqrt(double);
void sqrt(float);
```

它们都会被等同特化，因为它们中的任何一个都可以使用一个 float、double 或 real 来进行调用（如 2.3.2 节所讨论过的，尽管它是有损转换，但感觉是自相矛盾）。

也有可能出现没有一个函数会≤其他的函数，在这种情况下，我们说 foo_1 和 foo_2 是“无

① 翻破词典也找不到 math-fu 一词，它其实是演变自“kung fu”（功夫），故这里将其译作“数学功底”。——译者注

自在读书 www.zizidiary.com

序的”[①]。这种情况很多，例如：

```
// 两个无序函数：都不使用另一个函数的参数类型进行调用
void print(double);
void print(string);
```

最有趣的情况是，正好在 $foo_1 \leqslant foo_2$ 和 $foo_2 \leqslant foo_1$ 之中有一个为真——以第一种情况为例，在这种情形里，我们就说 foo_1 相比 foo_2 “不够特化”（less specialized）。即是说：

```
// 两个有序函数：write(double)没write(int)特化，
//       因为前者可以使用一个 int 来进行调用，
//       但后者不能使用一个 double 来调用
void write(double);
void write(int);
```

在重载调用 foo(arg1, ..., argn) 时，D 语言会运用这种偏序，执行下面的简单算法作出决定。

（1）如果有一个精确的匹配（拥有与实参相同类型和相同数量的形参），那么就选择它。

（2）如果没有其他重载存在时，就选择一个可接受调用的候选集 $\{foo_1, \cdots, foo_k\}$。在这里，类型推导会推断其类型，而 if 子句也会被计算。

（3）如果该集合的大小为零，那么就产生一个“无匹配”的错误。

（4）如果所有函数都不在同一个模块里，那么就产生一个“试图跨模块重载”的错误。

（5）在此集合中消除所有没其他函数特化的函数，即只保留最特化的那个函数。

（6）如果剩下集合的大小大于 1，那么就产生一个“有歧义的调用”的错误。

（7）此集合的唯一元素会成为“赢家”。

内容就这些。现在来看第一个示例：

```
void transmogrify(uint) {}
void transmogrify(long) {}

unittest {
   transmogrify(42);            // 调用 transmogrify(uint)
}
```

这里没有精确匹配，两个函数都是可以使用的，因此偏序就会发挥作用。即是说，尽管这两个函数都可以接受调用，但第一个更特化，因此它会获得调用。不论好坏，int 都会自动转换成 uint。现在我们再将一个泛型函数加入进来：

```
// 同上，外加...
void transmogrify(T)(T value) {}

unittest {
   transmogrify(42);            // 仍会调用 transmogrify(uint)
   transmogrify("hello");       // 会调用 transmogrify(T), T=string
   transmogrify(1.1);           // 会调用 transmogrify(T), T=double
}
```

现在，当 transmogrify(int) 与泛型函数 transmogrify(T)(T) 进行顺序比较时会发生什么事呢？不错，即使在比较顺序时已决定了 T = uint，T 也不会被替换成 uint，而是保留其所有泛型。transmogrify(int) 能接受有些随意的类型 T 吗？不能。transmogrify(T)(T)

① 这种情况正好是“偏序”（partial ordering）里的“偏”。在一个全序（例如，针对实数的“≤”）里，就没有无序元素。

自在读书 www.zizidiary.com

能授受 int 吗？当然可以。遵从 transmogrify(T)(T)没 transmogrify(int)那么特化的原则，因此它会从候选集中排除掉。所以非泛型函数通常比泛型函数的优先级更高，即使非泛型函数需要进行隐式类型转换也一样。

5.5.2　跨模块重载

在前面的重载算法里的第 4 步值得特别注意。假设稍微调整一下示例：使用 uint 和 long 来进行重载，另外再多加入几个文件：

```
// 在模块 calvin.d 里
void transmogrify(long) { ... }

// 在模块 hobbes.d 里
void transmogrify(uint) { ... }

// 模块 client.d
import calvin, hobbes;
unittest {
   transmogrify(42);
}
```

在 hobbes.d 里的 transmogrify(uint)重载是“更为特化的”。不过编译器会拒绝调用它，认为这里存在歧义。D 语言是坚决反对跨不同模块的重载的。如果允许这种重载的话，调用的含义就会依赖于被包含的各个模块间的相互作用（而在一般情况下也会有许多模块、许多重载以及更为复杂的调用卷入进来）。想象一下，往工作程序中添加一个新的 import，并以不可预知的方式更改它的行为，这会是什么样子！相反，如果允许跨模块重载，加载在代码阅读者身上的负担将大幅度增加：现在，为了找出一个调用跑到了哪里，他们不是要去了解一个模块所包含的内容，而是要了解所有包括进来的模块所包含的内容，因为其中任何一个模块都有可能会定义一个更好的匹配。更糟的是，如果顶层声明的顺序很重要，那么调用 transmogrify(5)实际上就可能根据其在文件里的位置而终止调用不同的函数。这会导致无穷尽的问题，因为它本质上意味着一段代码的读者在头脑里必须要随时大量移动上下文。

一个模块可以定义一大堆重载，让它们实现针对各种不同类型的功能。另一个模块则不能使用自己对该功能的附加操作进行干涉。不过，第二个模块可以定义它们自己的重载群。只要在一个模块里的函数不进行“劫持”调用（否则会转到另一个模块里的函数上去），就不会出现歧义。关于是否存在冲突的决定是以每一次的调用为基础进行的。假设：

```
// 在模块 calvin.d 里
void transmogrify(long) { ... }
void transmogrify(uint) { ... }

// 在模块 hobbes.d 里
void transmogrify(double) { ... }

// 在模块 susie.d 里
void transmogrify(int[]) { ... }
void transmogrify(string) { ... }

// 模块 client.d
```

自在读书
www.zizidiary.com

```
import calvin, hobbes, susie;

unittest {
    transmogrify(5);              // 出错！跨模块重载跨越了 calvin 和 hobbes
    calvin.transmogrify(5);       // 正确，显式调用 calvin.transmogrify(uint)
    transmogrify(5.5);            // 正确，只有 hobbes 能接受它
    transmogrify("hi");           // 来自 Susie 的问候
}
```

Calvin、Hobbes 和 Susie 以一种很有意思的方式进行交互。请注意：歧义粒度有多么的细。事实上，在第一个调用里，calvin.d 和 hobbes.d 之间有一个冲突，因为它没有体现模块的互不相容——第三个调用仍然通过了，因为在其他模块里没有函数能够接受它。最后，susie.d 定义了它自己的重载，并且未与其他两个模块出现冲突（与同名的漫画人物不太一样）。

重载指导　每当模块之间存在歧义时，你有两种主要方法可用来引导重载发挥作用。一种是使用模块名称来限定函数名，如第二个调用 calvin.transmogrify(5)。这样做就可以限制只在 calvin.d 里进行查找。在该模块内部，重载规则仍起作用。还有一种更透明的方式，即对有问题的符号使用本地别名，如下面这样：

```
// 在 calvin.d 内部
import hobbes;
alias hobbes.transmogrify transmogrify;
```

这个指令做了一件非常意义的事：它将模块 hobbes.d 里 transmogrify 的所有重载都引入到当前模块 calvin.d。因此如果 calvin.d 包含了上面的指令，那么它除了定义自己的重载外，还如同定义了在 hobbes.d 中定义的 transmogrify 的所有重载。calvin.d 变得非常不错——不管什么时候，只要是有关于调用 transmogrify 的重要决定，它都会民主地与 hobbes.dwhenever 进行协商。另外，如果 calvin.d 和 hobbes.d 遇到了问题并选择无视对方的存在，那么 client.d 会通过 calvin.transmogrify 和 hobbes.transmogrify 两个别名操作将所有重载考虑在内，从而仍然可以调用 transmogrify：

```
// 在 client.d 内部
alias calvin.transmogrify transmogrify;
alias hobbes.transmogrify transmogrify;
alias susie.transmogrify transmogrify;
```

现在，在 client.d 里产生的任何调用都会将在 calvin.d 和 hobbes.d 里定义的重载处理成仿佛是在 client.d 里定义的一样。

5.6　高阶函数、函数字面量

到目前为止，我们已知道如何在一个分割里查找一个元素或一个分割。然而，查找并不总是与搜索一个给定的项目相关。假设有这么一项任务：在一个数字数组里查找第一个负数元素。无论如何，我们的 find 库都无法解决这个问题。

从根本上讲，find 似乎是要满足某种布尔条件——谓词（predicate）。到目前为止，谓词一直都是针对一个给定的值使用==进行比较。更灵活的 find 应该能接收来自用户的谓词，并围绕着它装配出线性搜索逻辑。如果 find 可以积聚这种力量，那么它就会成为一个高阶函数，

自在读书 www.zizidiary.com

即可以将其他函数作为一个函数的参数。这是一种非常强大和实用的方法，因为高阶函数可以通过它们的参数所提供的功能复合出自己的功能，从而实现一系列简单函数不可能具有的行为。为了让 find 接受一个谓词，我们可以使用一个“别名参数”（alias parameter）。

```
T[] find(alias pred, T)(T[] input)
   if (is(typeof(pred(input[0])) == bool))
{
   for (; input.length > 0; input = input[1 .. $]) {
      if (pred(input[0])) break;
   }
   return input;
}
```

这个 find 的新重载除了只接受一个“传统”参数外，还添加了一个神秘的 alias pred 参数。一个别名参数可以匹配任何参数：值、类型、函数名——所有可用符号表示的内容。现在就来看看如何调用 find 的这个重载。

```
unittest {
   int[] a = [ 1, 2, 3, 4, -5, 3, -4 ];
   // 查找第一个负数
   auto b = find!(function bool(int x) { return x < 0; })(a);
   assert(b == [-5, 3, -4]);
}
```

这次 find 会接受两个参数列表。第一个列表是通过语法!(...)来进行区分，并由泛型参数构成。第二个列表是由传统的参数构成。请注意，尽管 find 声明了两个泛型参数（alias pred 和 T），而调用代码却只指定一个。这是因为推导过程会像往常一样，使用绑定 T = int。到目前为止，在所有 find 的应用里，我们都未指定过任何泛型参数，这是因为编译器会为我们推断出它来。这一次，由于无法推断谓词，因此我们将它指定为一个函数字面量：

```
function bool(int x) { return x < 0; }
```

这里的 function 是一个关键字，而其余部分则是常规的函数定义，只是没有名字而已。

事实证明函数字面量（也被称作匿名函数或 λ 函数）在很多情形下都是非常有用的，但它们的语法稍显繁重。上面使用的字面量有 41 个字符那么长，其中只有大约 5 个是干实际工作的。为帮上一点忙，D 语言允许你对这个语法进行一些修剪。第一种快捷方法是同时或单个消除返回类型和参数类型。编译器很聪明，它能根据其特有的定义将这些内容推断出来，而匿名函数的函数体还留在那里。

```
auto b = find!(function(x) { return x < 0; })(a);
```

第二种快捷方法就是将关键字 function 本身去除掉。你也可以将这两种快捷方法像下面那样进行合并，这会形成一个相当简洁的记号：

```
auto b = find!((x) { return x < 0; })(a);
```

很容易就能识别出它是在进行初始化，这个只需要花几秒钟的时间。

函数字面量与委托字面量

λ 函数功能有一个很重要的要求，即允许 λ 函数访问定义它的那个上下文。假设有一个稍微调整后的调用：

自在读书 www.zizidiary.com

```
unittest {
    int[] a = [ 1, 2, 3, 4, -5, 3, -4 ];
    int z = -2;
    // 查找第一个小于 z 的数
    auto b = find!((x) { return x < z; })(a);
    assert(b == a[4 .. $]);
}
```

这个调整后的示例是可以工作的，很明显，它回答了上面的问题。不过，出于好玩，我们试着在字面量里加上 function，这段代码神奇地停止了工作！

```
auto b = find!(function(x) { return x < z; })(a);
// 错误！函数不能访问调用者函数的帧！
```

发生什么啦？关于帧的抱怨是怎么回事？很显然，这里一定有某种潜在机制，函数字面量通过它可以访问到 z——它是不可能乱撞访问到的。这种机制被编码成一个隐藏参数——被称做“帧指针”（frame pointer），而字面量则会接受它。编译器会使用这个帧指针来连接对外部变量（如 z）的访问。然而，一个未使用任何局部变量的函数字面量并不需要这个额外的参数。D 语言属于静态类型的语言，它必须要将这两者区分开来，而且事实上它也做到了这一点。除了函数（function）字面量，还有委托（delegate）字面量——它可以像下面这样进行创建：

```
unittest {
    int z = 3;
    auto b = find!(delegate(x) { return x < z; })(a); // 正确
}
```

委托可以访问这个封闭帧，而函数不能。如果在字面量里，function 和 delegate 都不存在，那么编译器会自动检测需要哪一个。类型推断机制再次派上用场，它会生成最简洁、最方便的代码，同时也会自动把事情处理好。

```
auto f = (int i) {};
assert(is(f == function));
```

5.7　嵌套函数

现在我们可以使用任意函数字面量来调用 find，而且相当优雅。不过，如果该字面量变得特别大或者想要重用它，那么在调用它的地方（也有可能会多次）编写其函数体就会变得很笨拙。我们想用一个命名函数来调用 find，不想用匿名方式进行调用。此外，我们又想保留在需要时可以访问局部变量的权利。对于这种行为以及其他更多行为，D 语言可以使用嵌套函数。

嵌套函数的定义表面上跟普通函数完全一样，不同之处在于它是出现在另一个函数的内部。假设：

```
void transmogrify(int[] input, int z) {
    // 嵌套函数
    bool isTransmogrifiable(int x) {
        if (x == 42) {
            throw new Exception("42 cannot be transmogrified");
        }
        return x < z;
    }
    // 在输入里查找第一个 transmogrifiable 元素
    input = find!(isTransmogrifiable)(input);
```

自在读书 www.zizidiary.com

```
    ...
    // 再次查找
    input = find!(isTransmogrifiable)(input);
    ...
}
```

嵌套函数在很多不同情形下都很有用处。尽管所有常规函数不能做的它们也不能做，但嵌套函数增强了便利性和模块化程度，因为它们就位于使用它们的函数内部，并且它们可以紧密地访问到拥有嵌套函数的那个函数的上下文。最后一点优势特别突出——在上面的示例中，如果不使用嵌套函数，那么访问 z 就可能会存在更多的问题。

嵌套函数 isTransmogrifiable 使用了相同的技巧（一个隐藏参数）来获得对其上层栈帧的访问，特别是对 z 的访问。有时你可能并不想要发生这种事情，那么就只需将 isTransmogrifiable 当成一个完完全全的常规函数即可，而其定义还是可以保留在 transmogrify 内部。为达到这个效果，只需在 isTransmogrifiable 的定义里简单地添加一个 static（还需要什么吗？）：

```
void transmogrify(int[] input, int z) {
    static int w = 42;
    // 嵌套常规函数
    static bool isTransmogrifiable(int x) {
        if (x == 42) {
            throw new Exception("42 cannot be transmogrified");
        }
        return x < w; // 访问 z 就会出错
    }
    ...
}
```

借助于 static，isTransmogrifiable 就只能访问在 transmogrify 内部定义的模块级数组和静态数据（如示例里的 w）。任何临时数据（如函数参数或非静态变量）都不可访问（不过它们当然可以显式地利用传递）。

5.8　闭包

正如前面提到的，alias 完全就是一种符号技巧。它所做的就是让一个符号表示另一个符号所代表的相同内容。在上面的示例中，pred 并不是真实的值，就像任何函数名也不是一个值一样，你不能给 pred 赋值。如果想创建一个函数数组（如一系列命令），那么 alias 能起到不少作用。很显然这里还需要外加一些东西，它要有能力拥有一个实实在在的函数对象——它可以被存储和获取，与 C 语言里的函数指针非常相像。

例如，有这样一种要求：“给定 T 类型的值 x，然后返回一个函数——它能在 T 类型的数组里找到第一个等于 x 的值。”这种精炼的间接定义就是典型的高阶函数：你不是去做具体的事情，你只是返回某个内容，它会去负责完成。所以我们需要写一个函数，它会依次（请注意这里）返回一个函数，而这个函数接着（我们在不断往前翻滚!）会接受一个 T[]，并返回一个 T[]。因此，最终返回的类型为 T[] delegate(T[])。为什么是 delegate 而不是 function 呢？正如前面讨论过的，除了访问其参数外，delegate 还可以访问自己的状态；而 function 则不能。而我们的函数一定会拥有某种状态，因为它必须要保存 x 的值。

自在读书 www.zizidiary.com

这一点非常重要，因此有必要强调一下。假设将类型 T[] function(T[]) 当做一个函数的纯地址：一个机器字。那么，这个函数就只能访问其参数和程序的全局变量。如果你有两个指向同一函数的指针，并向它们传递同一个参数，那么它们将会访问到同一个程序状态。所有曾经处理过 C 语言的回调函数（如窗口系统或线程启动）的人都知道，这是个老大难问题：函数指针不能访问私有状态。C 语言的回调方式通常是使用一个 void*（一个无类型的地址），它会负责将状态信息四处传递。其他回调系统（如旧的、变化无常的 MFC 库）会在一个全局关联数组里存储额外的状态；但另外一些回调系统，如 Active Template Library（ATL），会使用汇编代码动态创建新的函数。跨平台的 Qt 库使用了一种被称作信号/插槽（signals/slot）的高级机制来实现同样的效果。无论在哪里与 C 语言的回调进行交互，都会涉及某种允许回调函数去访问状态的解决方案，这可不是一个简单的问题。

使用 delegate，所有这些问题都会消失殆尽。委托通过牺牲一点空间达到了这一要求：一个委托会保留一个函数指针和一个函数环境指针。这可能是大了点，有时还会更慢，但它同时也是异常地强大。因此，在你个人的设计中，应该多使用 delegate 对象，而不是 function 对象。当然，当通过回调函数与 C 语言进行交换时，function 也是不可替代的。

要说的就这些，现在我们来尝试编写一个新的 finder 函数。请记住，我们需要返回一个 T[] delegate(T[])。

```
import std.algorithm

T[] delegate(T[]) finder(T)(T x)
    if (is(typeof(x == x) == bool))
{
    return delegate(T[] a) { return find(a, x); };
}

unittest {
    auto d = finder(5);
    assert(d([1, 3, 5, 7, 9]) == [ 5, 7, 9 ]);
    d = finder(10);
    assert(d([1, 3, 5, 7, 9]) == []);
}
```

没错，对于初次见到它的人来说，像这种两个 return 语句被绑定到同一行的情形看起来的确很怪异，不过在当初许多高阶函数看起来也是很怪异的。现在逐行分析一下。finder 会被类型 T 参数化，它会接受一个 T，并返回一个 T[] delegate(T[])；finder 会将 T 用于条件验证：你必须要比较两个 T 类型的值，并得到一个 bool 类型。同样地，简单明了的比较 x == x 只适用于类型，而不是适合于特定的值。接着，finder 会通过返回一个委托字面量巧妙地履行自己的职责。这个字面量的函数体很短，它只是调用了我们之前定义的那个 find，而这个函数会负责履行契约。这个返回的委托便叫做“闭包”（closure）。

使用方法如预想的一样：调用 finder 会返回一个委托，在稍后可以调用它或再对它进行赋值。在这个单元测试里定义的变量 d 的类型为 T[] delegate(T[])，但因为有关键字 auto，我们在这里没有必要显式书写这个类型。老实说，auto 事实上也可以作为定义 finder 的一种快捷方法。这里提供所有类型的目的是为训练和讲解提供帮助。一个相当简短的 finder 的定义看起来就像这个样子：

```
auto finder(T)(T x) if (is(typeof(x == x) == bool)) {
    return (T[] a) { return find(a, x); };
}
```

自在读书 www.zizidiary.com

请注意：auto 可用做函数的结果类型，也可以用做省略的 delegate 关键字。编译器都会毫无问题地负责它。不过，委托参数前面的 T[] 是必须要指定的。这是因为编译器必须要有一个起始点才能让 auto 的这个“魔法”生效：该委托返回的类型可以根据 find(x) 的类型进行推断，而它又会根据 a 和 x 的类型进行推断。基于这点，这个委托表的类型就会被推断为 T[] delegate(T[])，它也是 finder 返回的类型。如果不知道 a 的类型，则这一连串的推理都没法进行下去。

它能不能工作

单元测试提供了探测 finder 行为的能力，仅这一点当然还不能证明它就可以正常地工作。在其背后还潜在一个很高级的问题：finder 返回的委托使用了 x。在 finder 返回之后，x 会在哪里呢？事实上，假定 D 语言调用函数是使用常规的调用栈，那么这个问题就暗示会发生某些非常危险的事：调用者调用 finder，x 会存放到调用栈上；finder 返回，这个栈会被弹回其调用之前的位置……即表示 finder 返回的委托会访问一个已消失的栈位置！

局部环境的持久化（在我们的情况里，环境只由 x 构成，但它可能是任意大的）是闭包实现中的一个典型问题，每一种支持闭包的语言都必须以某种方式对它进行说明。D 语言使用这样一种方法[①]：通常，所有调用都会使用一个简单栈。当编译器检测到闭包存在时，它会自动将用到的环境复制到基于堆的内存上，并且告知委托使用那个在堆上分配的数据。这个堆会被执行垃圾回收。

这种做法的不足之处在于，每一次调用 finder 都会引发一个内存分配请求。然而，闭包具有很丰富的表现力，它开启了一系列很有意思的编程范型。因此，在大部分情况下，这种开销是完全可以接受的。

5.9　超越数组、范围、伪成员

在 5.3 节的末尾，我们神神秘秘地说到：“find 过于通用，同时又不够通用。”然后，我们看到了 find 是如何“过于通用”的，并且通过限制其接受的类型来将它修复。现在是时候找出为什么 find 仍然“不够通用”的原因了。

线性搜索的本质是什么呢？你一个接一个元素地遍历被搜索对象以查找一个特定的值，或满足一个谓词的值。问题在于，到目前为止我们一直都只忙于处理连续数组（如 T[] 那样的分割），还从未接触过与线性搜索相关的概念（它只与从事这个领域的机制有关）。我们将自己限制在 T[] 上面，便可以让 find 访问大量其他类型的数据结构，这些结构都拥有各种可以线性搜索的拓扑结构。例如，如果有一种语言让 find 的某个方法拥有像类似 Array 那样的类型，那么它肯定被你蔑视。这并不是说你不能用这种语言来完成工作，而是说你在这个问题上肯定会很忙，而实际上你却没必要这样忙。

是时候清除障碍，重新评估一下我们的 find 基本实现。方便起见，现将代码复制如下：

```
T[] find(T)(T[] haystack, T needle) {
    while (haystack.length > 0 && haystack[0] != needle) {
```

① 类似于 ML 和其他函数式语言实现的方法。

自在读书 www.zizidiary.com

```
        haystack = haystack[1 .. $];
    }
    return haystack;
}
```

对于 haystack，我们使用的原语是什么呢？它们各自的含义又是什么呢？

(1) haystack.length > 0 用于说明在 haystack 里还剩有元素。

(2) haystack[0]用于访问 haystack 的第一个元素。

(3) haystack = haystack[1 .. $]用于消除 haystack 的第一个元素。

数组实现这些操作的特定方式并不容易推广到其他容器上。例如，对于一个单链表通过计算 haystack.length > 0 来询问是否还有元素，这个可要荣获达尔文奖[①]了。除非该列表能始终缓存其的长度（问题远不止这个），计算一个列表的长度所需时间是与列表实际的长度成比例的，而快速查看列表的头节点只需要几条机器指令。使用索引方式访问列表同样是个“亏本买卖”。因此，我们要将这些原语提炼成 3 个命名函数，并忽略掉 haystack 的类型，以找出它们的实现。这些原语的语法可能如下。

(1) haystack.empty 用于测试 haystack 是否已完成。

(2) haystack.front 用于获取 haystack 的第一个元素。

(3) haystack.popFront()用于消除 haystack 的第一个元素。

请注意，前两个操作并不会修改 haystack。因此，它们未使用圆括号，而第三个则影响了 haystack，这个通过使用()的方法在语法上进行强调。现在我们使用这个新语法重新定义 find：

```
R find(R, T)(R haystack, T needle)
    if (is(typeof(haystack.front != needle) == bool))
{
    while (!haystack.empty && haystack.front != needle) {
        haystack.popFront();
    }
    return haystack;
}
```

在这个包容性定义的作用下，情况好多了，但单元测试失败还是令人有些不快。没错，当然可以进行修正：内建的分割类型 T[]并不知道我们突然会受到启发去选择一套新的原语，关于这些原语暂且随意给它们一个名字，如 empty、front 和 popFront。我们需要为所有的 T[]定义这些操作。当然，它们都会有一个无关紧要的实现，但我们需要这些内容，以便让我们设计好的抽象可以随同我们一开始就使用的类型再一次工作起来。

5.9.1　伪成员和@property 属性

这里有一个语法问题，即函数调用到目前为止看起来像 fun(argument)这个样子；而现在我们想让定义出的调用看起来像 argument.fun()和 argument.fun 这个样子。后面两种语法分别叫做“方法调用语法”和“属性访问语法”。在下一章会学习到它们。对于自定义类型，

① 达尔文奖（Darwin Awards），这是一个带半开玩笑性质的奖项。具体解释可参考 http://en.wikipedia.org/wiki/Darwin_Awards。——译者注

自在读书 www.zizidiary.com

它们很容易进行定义，但 T[] 是一个内建的类型，该怎么做呢？

D 语言认为这是一个纯语法的问题，因此允许使用伪成员（pseudo member）记号：如果看到 a.fun(b, c, d)，但 fun 不是 a 的类型所含有的成员，那么 D 语言会将它重写成 fun(a, b, c, d) 并再次进行尝试。绝不会选择到反向路径，不过如果写成 fun(a, b, c, d)，而它又没什么意义的话，那么也不会尝试 a.fun(b, c, d)。伪方法的目的在于允许你使用“发送消息给对象”的语法来调用常规的函数，而这对于我们中的许多人来说是再熟悉不过的。不再多说，一起来为内建数组实现 empty、front 和 popFront。只需 3 行就能完成：

```
@property bool empty(T)(T[] a) { return a.length == 0; }
@property ref T front(T)(T[] a) { return a[0]; }
void popFront(T)(ref T[] a) { a = a[1 .. $]; }
```

记号@property 引入的属性（attribute）被称作特性（property）。这种总是由@引入的属性，是一种简单的修饰符，它们用于指定所定义符号的某种特征。有些属性是编译器公认的，有些则是程序员自己定义和使用的。特别是“特性”，它是编译器公认的，并传达出这样一个信息：与这个属性相关的那个函数在被调用时不用带上末尾的“()”。

也请注意，有两个地方使用了 ref（参见 5.2.1 节）。一个是 front 的返回类型里，其目的在于允许修改数组里的元素。另外 popFront 也使用了 ref，其目的在于确保其更改会直接作用到那个分割上。

借助这 3 个简单的定义，修改后的 find 编译和运行都没问题，完全令人满意。我们泛化了 find，让它对任何定义了 empty、front 和 popFront 的类型都可以工作。接着回到起点，让通用版本对于具体的情况（开始时，是它激发了泛化操作）也能够工作。如果这 3 个原语是内联的，那么这个泛化的 find 就能保持与前一种实现（它只用来处理分割）一样的效率，而该实现只能处理分割。

现在，如果 empty、front 和 popFront 都只对定义 find 有用，那么这种抽象则明显还不够让人印象深刻。没错，我们是从 find 解脱了，但是当我们定义另一个函数时，这个 empty-front-popFront 三件套是否还有别的用处吗？或者我们必须要使用某些不一样的原语从头开始吗？庆幸的是，大量经验表明：这里有一个非常基本的想法，即可以将访问传递给一个数据集合。这个想法很有用，《设计模式》（*Design Patterns*）一书将它定义为迭代器模式[27]；C++ 的 STL[51]将这个想法做了进一步的改进，定义了一个迭代器的概念层：输入（input）迭代器、前进（forward）迭代器、双向（bidirectional）迭代器和随机访问（random-access）迭代器。

在 D 语言的术语里，允许遍历元素集合的抽象数据类型叫做“范围”（range）。迭代器（iterator）也是一个不错的可供选择的名字，但已经存在的各种不同的库已让这一术语具有了不同的含义。如果在 D 语言里继续使用这一术语，可能会导致混淆。D 语言的“范围”比 STL 的迭代器更近似于迭代器模式（D 语言的“范围”模型相当于一对 STL 迭代器模型），它还继承了 STL 的分类方法。empty-front-popFront 三件套特意定义了一个“输入范围”（input range），因此我们对优秀 find 函数的不懈追求显示了线性搜索和输入范围之间密不可分的关系：没有输入范围，就不能进行任何线性搜索；无缘无故地向集合请求超出输入范围能力之外的功能，将会出现错误（例如，根本不需要支持索引访问的数组）。在标准库的 ststd.algorithm 模块里，可以找到

自在读书 www.zizidiary.com

一个几乎一样的 find 实现。

5.9.2 reduce——不只是证法

来挑战一下只使用输入范围怎么样？具体做法就是：定义一个函数 reduce，假定有一个输入范围 r、一个操作 fun 和一个种子 x，那么它会针对 r 里的第一个元素 e 重复地计算 x = fun(x, e)，然后返回 x。这个高阶函数 reduce 非常强大，因为它表示各种有趣的累计运算。有许多具备高阶函数能力的语言将 reduce 定义成一个主要工具，可能使用的名字为：accumulate、compress、inject 或者 foldl。现在开始定义 reduce，按照测试驱动开发的思想，同时会带上几个单元测试：

```
unittest {
    int[] r = [ 10, 14, 3, 5, 23 ];
    // 计算所有元素的总和
    int sum = reduce!((a, b) { return a + b; })(0, r);
    assert(sum == 55);
    // 计算最小值
    int min = reduce!((a, b) { return a < b ? a : b; })(r[0], r);
    assert(min == 3);
}
```

正如你所看到的，reduce 非常灵活和有用，当然，前提是我们将那个还不存在的次要细节忽略掉了。现在就来实现 reduce，这样它就能满足上面的测试了。我们现在有了从头开始编写真正工业级 reduce 的知识：从 5.3 节，我们了解到如何给函数传递类型参数；5.4 节教会了我们如何限制 reduce，以便让它只接受合理的参数；在 5.6 节，我们看到了函数字面量是如何被当做 alias 参数传递给函数的，我们刚才又将注意力集中到了漂亮、简单的输入范围接口上。

```
V reduce(alias fun, V, R)(V x, R range)
    if (is(typeof(x = fun(x, range.front)))
        && is(typeof(range.empty) == bool)
        && is(typeof(range.popFront())))
{
    for (; !range.empty; range.popFront()) {
        x = fun(x, range.front);
    }
    return x;
}
```

编译它，然后运行 unittest，最后所有测试都顺利通过。不过，如果约束不是像实现本身那样笨重的话，reduce 的定义将会更可爱。此外，没有人愿意去编写那个冗长乏味的测试——它用于确保 R 是一个输入范围。这个冗长的约束是稍微有些重复。幸运的是，在标准模块 std.range 里，我们有一个包装精良的用于范围的测试，它简化了 reduce 的实现：

```
import std.range;

V reduce(alias fun, V, R)(V x, R range)
    if (isInputRange!R && is(typeof(x = fun(x, range.front))))
{
    for (; !range.empty; range.popFront()) {
        x = fun(x, range.front);
    }
    return x;
}
```

自在读书 www.zizidiary.com

这个明显更受人欢迎。利用 reduce，不仅可以进行总和与最小值的计算，也可以计算其他聚集函数，如最靠近指定数的数、最大绝对值、标准偏差。标准库在 std.algorithm 里定义的 reduce 与上面那个几乎相同，不同之处在于它是接受多个函数来进行计算——在计算多个聚集函数时，它的效率很高，因为它对输入只做了一次遍历操作。

5.10　可变参数函数

在第 1 页里的那个传统“Hello, world!”程序使用了标准库函数 writeln 来将欢迎信息输出到标准输出设备。关于 writeln 有一个很有趣的细节，即它可以接受任意数量和类型的参数。在 D 语言里，有好几种方法可用于定义可变参数函数（variadic function），以满足不同的需求。我们先从最简单的开始。

5.10.1　同构可变参数函数

同构可变参数函数可以接受任意数量相同类型的参数，其定义形式如下：

```
import std.algorithm, std.array;

// 计算一组数据的平均值，
//     可以直接传递，也可以通过数组传递。
double average(double[] values...) {
    if (values.empty) {
        throw new Exception("Average of zero elements is undefined");
    }
    return reduce!((a, b) { return a + b; })(0.0, values)
        / values.length;
}

unittest {
    assert(average(0) == 0);
    assert(average(1, 2) == 1.5);
    assert(average(1, 2, 3) == 2);
    // 传递数组和分割也同样工作
    double[] v = [1, 2, 3];
    assert(average(v) == 2);
}
```

请注意，这里再次使用 reduce 恰到好处。average 最明显的特征就是在参数 values 的后面紧跟有一个省略号“...”，同时 values 是分割类型。如果它不存在，或者如果 values 不是 average 的最后一个参数，那么省略号就会是一个错误。

如果你使用一个 double 分割来调用 average，那么它跟平常没什么两样，如那个单元测试的最后一行所示。但是多亏有省略号，你也可以使用任意数量的参数来调用 average，前提是每一个都能转换为 double。编译器会自动排列分割里的所有参数，并将它传递给 average。

你可能会认为，这个特性实际就是一颗语法糖（syntactic sugar），它只是将 average(a,b,c) 重写成 average([a,b,c])。但正是因为有了这种调用语法，同构可变参数函数才能在其作用域里重载其他函数。例如：

自在读书 www.zizidiary.com

```
// 为了参数
double average() {}
double average(double) {}
// 同构可变参数函数
double average(double[] values...) { /* 同上 */ ... }
unittest {
   average();          // 错误！重载函数调用存在歧义！
}
```

最开始的两个 average 重载与使用 0 个和 1 个参数来调用 average 的可变参数版本会产生歧义。显式地将一个分割传递给 average 可以消除这种歧义，如 average([1,2])。

对相同的分割类型，如果非可变参数函数和可变参数函数都适合，那么在调用过程中实际传递的是分割时，则会优先选择非可变参数函数：

```
import std.stdio;

void average(double[]) { writeln("non-variadic"); }
void average(double[]...) { writeln("variadic"); }
void main() {
   average(1, 2, 3);    // 输出 "variadic"
   average([1, 2, 3]); // 输出 "non-variadic"
}
```

5.10.2　异构可变参数函数

现在回到 writeln，很显然它一定做了某种不同于 average 所做的事，因为 writeln 可以接受不同类型的参数。为匹配任意数量和类型的参数，你应该使用异构可变参数函数，其定义样式如下：

```
import std.conv;

void writeln(T...)(T args) {
   foreach (arg; args) {
      stdout.rawWrite(to!string(arg));
   }
   stdout.rawWrite('\n');
   stdout.flush();
}
```

这个实现有点粗糙，且效率不高，但它可以工作。在 writeln 内部，T 是一个“参数类型元组”（parameter type tuple），这是一种可以将多种类型包裹在一起的类型；而 args 是一个“参数元组”（parameter tuple）。foreach 语句能检测出 args 就一个元组类型，因此它会生成与平常的 foreach 调用（如用于迭代数组的调用）截然不同的代码。假设有下面的调用：

```
writeln("Writing integer: ", 42, " and array: ", [ 1, 2, 3 ]);
```

对于这个调用，由 foreach 生成的代码有点像这个样子：

```
// 生成代码的大致样子
void writeln(string a0, int a1, string a2, int[new] a3) {
   stdout.rawWrite(to!string(a0));
   stdout.rawWrite(to!string(a1));
   stdout.rawWrite(to!string(a2));
   stdout.rawWrite(to!string(a3));
   stdout.rawWrite('\n');
   stdout.flush();
}
```

自在读书 www.zizidiary.com

模块 std.conv 定义的 to!string 可用于所有类型（包括字符串自身，对于这种情况，to!string 会成为恒等函数），因此这个函数的工作方式是：将每个参数都转换为一个字符串，然后将它作为原始字节输出到标准输出设备。

你不需要使用 foreach 来访问参数元组里的类型或值。如果 n 是一个编译时整数常量，那么 T[n]会得到参数元组里第 n 个类型，而 args[n]会得到参数元组里的第 n 个值。为获得参数个数，可以使用 T.length 或 args.length（两者都是编译时常量）。如果注意到它与数组的相似之处，发现 T[$ - 1]可以访问 T 里的最后一个类型时（args[$ - 1]是 args 里最后一个值的别名），就一点也不奇怪了。例如：

```
import std.stdio;

void testing(T...)(T values) {
   writeln("Called with ", values.length, " arguments.");
   // 访问每一个索引和值
   foreach (i, value; values) {
      writeln(i, ": ", typeid(T[i]), " ", value);
   }
}

void main() {
   testing(5, "hello", 4.2);
}
```

此程序会输出：

```
Called with 3 arguments.
0: int 5
1: immutable(char)[] hello
2: double 4.2
```

5.10.2.1　无名类型

writeln 函数做了太多特定的事情，因而无法通用——在末尾它总是会输出字符'\n'，并将流清空。我们想要根据基本的 write 函数（它只会依次输出每一个参数）来定义 writeln：

```
import std.conv;

void write(T...)(T args) {
   foreach (arg; args) {
      stdout.rawWrite(to!string(arg));
   }
}

void writeln(T...)(T args) {
   write(args, '\n');
   stdout.flush();
}
```

请注意 writeln 是怎样将 args 加上'\n'后转到 write 的。当转发一个参数元组时，它会自动扩展，这样调用 writeln(1, "2", 3)时，会有 4 个参数而非 3 个参数转到 write。这种行为有点不合常理，且令人费解，因为在 D 语言里几乎所有地方提到一个符号时就意味着有一个值。这里有一个示例，即使你心里有数也可能会大吃一惊：

```
void fun(T...)(T args) {
   gun(args);
}

void gun(T)(T value) {
```

自在读书 www.zizidiary.com

```
    writeln(value);
}

unittest {
    fun(1);       // 正确
    fun(1, 2.2); // 错误！不能找到接受两个参数的'fun'函数！
}
```

第一个调用顺利通过，但第二个调用却没有。之所以认为它会通过，是因为任何值都有一种类型，因而 args 也一定拥有某种类型，它可以被 gun 推断出来的。到底是怎么回事呢？

答案就是：事实上，所有值都有一种被编译器正确追踪的类型。罪魁祸首是 gun(args) 调用，因为只要参数元素以参数形式传递给函数，编译器都会自动将其扩展。因此，即使你写的是 gun(args)，编译器也总会将它扩展成 gun(args[0], args[1], ..., args[$ - 1])。在第二个调用里，它表示的是 gun(args[0], args[1])，即它会去查找带有两个参数的 gun 函数，结果没有找到——因此出错。

将调查再深入一点，让 fun 输出 args 的类型：

```
void fun(T...)(T args) {
    writeln(typeof(args).stringof);
}
```

操作 typeof 并不是一个函数调用。它只会得到 args 的类型，因此我们不必担心自动扩展的问题。任何类型的特性.stringof 都会得到它的名字，所以再次编译并运行这个程序。它会输出：

```
(int)
(int, double)
```

这样看起来编译器的确跟踪了参数元组的类型，并且使用一个字符串来表示它们。不过，不能显式定义一个参数元组——没有被称做(int,double)这样的类型：

```
// 没有效果
(int, double) value = (1, 4.2);
```

这是因为这些元组在这里是唯一的：它们是编译器内部使用但不能被表示的类型。你无法坐下来写出一种参数元组类型。对于这个问题，并没有能产生参数元组的字面量值（如果有的话，auto 便可以不再需要使用类型名）。

5.10.2.2　Tuple 类型和 tuple

没有名称的类型和没有字面量的值，对于寻求刺激的探索者来说是一种相当有意思的概念。但它带给实际工作中的程序员的损害却是极大的。幸运的是（终于幸运了一回），实际上它并不是一个限制，而更多的是一种精简语法的方法。你可以通过使用标准库的 Tuple 类型，以及带有 tuple 的参数元组值，单独地表示参数元组类型。这两者都可以在标准模块 std.typecons 里被找到。所以包含有一个 int 和一个 double 的参数元组可以写成下面的样子：

```
import std.typecons;
unittest {
    Tuple!(int, double) value = tuple(1, 4.2); // 哟！
}
```

同样地，tuple(1, 4.2)会返回一个类型为 Tuple!(int, double)的值：

自在读书 www.zizidiary.com

```
auto value = tuple(1, 4.2); // 哟哟!
```

像其他类型一样，Tuple!(int, double)作为一种类型，并不会自动进行扩展。因此，如果你想要将它扩展成它的组成成分，则必须显式地使用 Tuple 的 expand 特性。例如，我们可以将 fun/gun 程序拆散，然后将它重写成下面的样子：

```
import std.stdio, std.typecons;

void fun(T...)(T args) {
    // 创建一个元组，它会将所有参数包裹在一起
    gun(tuple(args));
}

void gun(T)(T value) {
    // 对元组进行扩展
    writeln(value.expand);
}

void main() {
    fun(1);       // 正确
    fun(1, 2.2); // 正确
}
```

请注意 fun 是如何将其所有的参数包裹成一个 Tuple，并将它传递给 gun 的。函数 gun 会将接收到的元组扩展成其包含的任何内容。表达式 value.expand 被自动重写成一个参数列表——它包含了你放入到那个 Tuple 里的任何内容。

Tuple 的实现有点巧妙，但使用方法任何程序员都可以获得的。检查其在标准库里的定义有助于加强练习。

5.11　函数属性

D 语言函数允许使用一些附带的“属性”（attribute）——这种特定的特征会向程序员和编译器通告函数的某些特性。函数的属性会被检查，因此为得到关于函数行为的重要信息，用户必须要做的是查看它的签名，以确认这种担保不仅仅是一种注释或一种惯例。

5.11.1　纯函数

函数的纯度（purity）是一个借用自数学的概念，它有着相当多的理论和实践优势。在 D 语言里，如果一个函数的唯一作用就是返回结果，并且其结果只依赖于它的参数，那么就认为它是“纯的”（pure）。

在经典数学里，所有函数都是纯的，因为经典数学不使用状态和突变。$\sqrt{2}$ 的结果是多少？没错，是 1.4142，就像昨天、明天或者任何时间点一样。可以这样说，即使是在人类发现根号、代数、数字之前，或者甚至是在人类欣赏数学之美之前，$\sqrt{2}$ 就拥有了这个值；而在很久之后宇宙因热耗尽而灭亡时，它也会保持同样的值。其数学结果是永恒的。

纯度对函数很有好处，至少有时是这样；而且像生活一样，还能提高品质。同样也像生活

自在读书 www.zizidiary.com

一样，函数的纯度并不容易实现。不过，至少还是可以遵从一些，过度的纯度也是很烦人的。在好的一面，纯函数更容易进行推理。要知道，你可以在任何时间调用它，只需检查其调用就可以了解到它应具有的作用是什么。你可以使用等效的函数调用来代替值，也可以使用值来代替等效的函数调用。要知道，纯函数里的错漏绝不会引起“遍地开花”的问题——这种错漏绝不会影响到函数的结果自身以外的东西。

同时，纯函数可以照字面的方式并行运行，因为它们不会跟程序的其他部分进行交互（它们的结果除外）。相比之下，频繁突变的函数在并行运行时，彼此之间会产生干扰。即使是按序列运行，它们发挥的作用很可能会莫名地依赖于调用它们的顺序。许多人都非常习惯这种做事的方式，考虑到这部分工作的难度，我们也只好作罢。不过如果一个应用程序的有些部分服从纯度，也是很新颖和有用的。

通过在函数定义的前面加上前缀 pure 便可以定义纯函数：

```
pure bool leapYear(uint y) {
    return (y % 4) == 0 && (y % 100 || (y % 400) == 0);
}
```

此函数的签名为：

```
pure bool leapYear(uint y);
```

它向用户保证 leapYear 不会向标准输出设备输出内容。此外，只通过查看其签名即可明白：不管是现在还是过去，leapYear(2020)都会返回相同的值。

编译器对 pure 也很敏锐，而实际上它会守护着 leapYear，以防止出现任何会导致 leapYear 无法保持纯净状态的操作。假设像下面这样进行更改：

```
pure bool leapYear(uint y) {
    auto result = (y % 4) == 0 && (y % 100 || (y % 400) == 0);
    if (result) writeln(y, " is a leap year!"); // 错误!
        // 不能从纯函数里调用不纯的函数!
    return result;
}
```

writeln 函数不是并且也不能是纯的。如果将它声明为纯的，则编译器会对这种假象进行矫正。编译器会确保纯函数不能调用不纯的函数。这就是为什么修改后的 leapYear 无法编译通过的原因。另一方面，编译器也能成功地对 daysInYear 这样的函数进行验证：

```
// 验证过的始终还是纯的
pure uint daysInYear(uint y) {
    return 365 + leapYear(y);
}
```

pure 就是要把它做得够纯

传统意义上，为了让程序可以宣称其纯度，函数式语言会要求绝对不会出现更改。D 语言将这种要求进行了放宽，即允许函数更改它们自己的私有状态和临时状态。这样，即使内部存在更改，但从函数外部看来仍然是纯净的。

现在我们看看这个放宽的操作是如何工作的。假设这里有一个比较天真的函数式 Fibonacci（斐波纳契）函数的实现：

自在读书 www.zizidiary.com

```
ulong fib(uint n) {
   return n < 2 ? n : fib(n - 1) + fib(n - 2);
}
```

没有哪位计算机科学老师会教出这样的 Fibonacci 实现。fib 需要花费指数级的时间才能完成计算，而且它除了忽视复杂性和计算成本外毫无任何意义——这是一种“草草了事”的态度和“鲁莽行事”[①]。要知道“指数级”有多糟吗？在一台当代机器上，fib(10)和 fib(20)所花费的时间可以忽略不计，但 fib(50)要花费 19 分钟。计算 fib(1000)所花费的时间很可能会超过人类存在的时间（它比 $\sqrt{2}$ 的计算方法可差远了）。

很好，那“更环保”的函数式 Fibonacci 实现是什么样子呢？

```
ulong fib(uint n) {
   ulong iter(uint i, ulong fib_1, ulong fib_2) {
      return i == n
         ? fib_2
         : iter(i + 1, fib_1 + fib_2, fib_1);
   }
   return iter(0, 1, 0);
}
```

修订后的版本在计算 fib(50)时所花费的时间是微乎其微的。现在这个实现花费的时间复杂度为 $O(n)$，而“尾部调用消除”（参见 1.4.2 节）技术会负责空间复杂度的事。值得一提的是，有些计算 Fibonacci 的算法其时间复杂度已达到 $O(\log n)$。

这里的问题在于新的 fib 有点暗淡失色。本质上，修订后的实现需要维持两个伪装成函数参数的状态变量，因此我们还可以进行进一步清理，编写出更简单的循环，这样 iter 就没必要存在了：

```
ulong fib(uint n) {
   ulong fib_1 = 1, fib_2 = 0;
   foreach (i; 0 .. n) {
      auto t = fib_1;
      fib_1 += fib_2;
      fib_2 = t;
   }
   return fib_2;
}
```

但是，这个不再是函数式的。看看在循环里发生的所有更改。一旦某一步弄错，我们就可能从数学纯度的“山峰”一下跌至不纯净的“万丈深渊”。

要是我们稍坐片刻，仔细想想，会发现迭代 fib 也不是那么不纯净。如果将它当做一个黑盒，那么对于一个给定的输入，fib 总是会输出相同的内容，即不管怎样，纯的始终还是纯的。事实上，它使用私有可变状态可能会让它在字面上没那么函数式，但在思想上它还是函数式的。仔细顺着这个思路，我们会得到一个非常有趣的结论：只要函数里的可变状态完全是临时的（即在栈上进行分配的）和私有的（即未通过引用传递到函数，因为这样做会污染到函数），那么便可以认为此函数是纯的。

D 语言对函数纯度的定义是：你可以在一个纯函数的实现里对状态进行更改，只要它是临时的和私有的即可。然后你可以将 pure 放置到这个函数的签名里，编译器也能顺利地编译它：

```
pure ulong fib(uint n) {
```

① 原文为 SUV driving，SUV（Sport Utility Vehicle）即指运动型汽车。这里将其译作“快速驾驶”，以此形容这个实现一味追求快速实现，而未做深入考虑。　——译者注

自在读书 www.zizidiary.com

```
    ... // 迭代实现
}
```

D 语言纯度的放宽方式是非常厉害的，因为这是个两全齐美的解决办法：坚固地保证了函数的纯度，同时在更改成为首选方法时也可以舒适地去实现。

5.11.2　函数属性 nothrow

给一个函数指定 nothrow 属性，它会传达出这样一条信息：该函数绝不会抛出异常。就像 pure 一样，nothrow 是编译时检查的。例如：

```
import std.stdio;

nothrow void tryLog(string msg) {
    try {
        stderr.writeln(msg);
    } catch (Exception) {
        // 忽略异常
    }
}
```

这个 tryLog 函数会试着对消息进行记录。如果有异常抛出，则安静地将其忽略掉。这样，tryLog 在代码的关键部分就变得很有用。在某些情况下，有些重要的事务之所以失败恰好就是因为日志消息无法写入，这是很悲哀的事情。拥有事务语义的代码需要批判地依赖于某些绝不会抛出异常的部分代码，而 nothrow 让你可以静态地确保这类情况。

nothrow 函数的语义检查会确保在这个函数里绝不会有异常泄漏出来。本质上，在这种函数里的任何语句都不会包含 throw（即调用的是另一个 nothrow 函数），或者这些语句是嵌入在一个 try 语句内部的——它会将异常吞掉。现在举例来说明第一种情况：

```
nothrow void sensitive(Widget w) {
    tryLog("Starting sensitive operation");
    try {
        w.mayThrow();
        tryLog("Sensitive operation succeeded");
    } catch (Exception) {
        tryLog("Sensitive operation failed");
    }
}
```

第一个 tryLog 调用没必要放置在 try 语句内，因为编译器已经知道它不会抛出异常。同样地，在 catch 子句里的调用也不需要通过额外的 try 语句来进行保护。

pure 和 nothrow 之间的关系是什么呢？表面看来，它们是完全正交的，但它们也可能存在某种程度的关联。至少在标准库里，许多数学函数既是 pure 同时也是 nothrow，如大多数超越函数（exp、sin、cos 等）。

5.12　编译时计算

常言说得好：好东西总是留给等到最后的人（或耐心读完的人）。作为最后一节，它要讨论

自在读书 www.zizidiary.com

D 语言的一个非常有趣的特性。特别之处在于：你不需要费劲学习便可以顺利地使用这个特性。

我们来看一个示例，它有点大，但很有意思。假设要定义完整的随机数生成器库。现在已存在许多随机数生成器，其中有快速的和被仔细研究过的“线性同余生成器”[35，3.2.1 节，10～26 页]。这种生成器有 3 个整型参数：模 m，满足 $m>0$；乘数 a，满足 $0<a<m$；并且增量 c[①]，满足 $0<c<m$。从任意一个种子 x_0（满足 $0\leqslant x_0<m$）开始，线性同余生成器使用下面的递推公式会产生一系列伪随机数：

$$x_{n+1}=(ax_n+c) \bmod m$$

这个算法的代码很简单——所有你需要的就是保留由 m、a、c 和 x_n 定义的状态，并定义一个 `getNext` 函数，让它可以将 x_n 更改成 x_n+1，然后返回它。

但这里有个小问题。并不是所有 m、a 和 c 的组合都能产生一个好的随机数生成器。首先，如果 $a=1$，且 $c=1$，那么生成器产生的序列为 0, 1, …, m–1, 0, 1, …, m–1, 0, 1, …不可否认这是很不随机的。

如果 a 和 c 的值再大一些，则这种比较明显的风险便可避免，但会出现一个隐性的问题：周期性。因为模操作符的关系，所生成的数字总是介于 0 和 m–1 之间，因此应尽可能让 m 更大一些（通常为 2 的幂，以匹配机器字的大小，这样模运算就不会有影响了）。真正的问题在于，在一段期间内生成的序列可能远小于 m。假设我们使用 `uint` 进行运算，并选择 $m=2^{32}$，甚至也不需要模运算。即让 $a=210$，$c=123$，以及将 x_0 设置成一个近似疯狂的值，如 1 780 588 661。现在运行此程序：

```
import std.stdio;

void main() {
   enum uint a = 210, c = 123, x0 = 1_780_588_661;
   auto x = x0;
   foreach (i; 0 .. 100) {
      x = a * x + c;
      writeln(x);
   }
}
```

数字的随机显示并不是“绚丽多彩”，我们看到的内容有些令人吃惊：

```
1     261464181
2     3367870581
3     2878185589
4     3123552373
5     3110969461
6     468557941
7     3907887221
8     317562997
9     2263720053
10    2934808693
11    2129502325
12    518889589
13    1592631413
14    3740115061
15    3740115061
16    3740115061
17    ...
```

这个生成器一开始很正常。至少在外行看来，它的随机数生成工作完成得很好。但在不超过 14

① $c=0$ 的情况也是允许的，不过相关的理论更难，因此这里我们只要求 $c>0$。

自在读书 www.zizidiary.com

个步骤之后，它便停滞在了一个固定点：通过其中一个奇怪的只有数学才有能力遇上的巧合，3740115061（过去和将来）恰好等于$(3740115061 \times 210 + 123) \bmod 2^{32}$。周期为 1，这是最坏的可能！

因此我们需要确保选择的 m、a 和 c 能让这个生成器拥有一个很大的周期。对此事的进一步调查，结果表示产生一个周期为 m（最大可能周期）的序列所需的条件如下。

（1）c 和 m 都是互质的。

（2）$a-1$ 可被 m 的所有素因子整除。

（3）如果 $a-1$ 是 4 的倍数，那么 m 也要是 4 的倍数。

c 和 m 的互质性可以轻易地通过比较它们针对 1 的最大公约数来进行检查。关于最大公约数的计算，我们可以使用欧几里得（Euclid）算法[①]：

```
// 实现 Euclid 算法
ulong gcd(ulong a, ulong b) {
   while (b) {
      auto t = b;
      b = a % b;
      a = t;
   }
   return a;
}
```

欧几里得算法就是使用减法代替模运算。模运算版本使用了更少的迭代，但对于今天的机器，操作符“%”执行起来依然相当缓慢，看来欧几里得早已想到了这一点。

第二个测试实现起来有点难度。我们可以先编写一个函数 factorize（它会返回一个数的所有素因子以及它们的乘方），然后再使用它，但 factorize 不是很有必要的。想要最简单又能工作的设计，简单的选择或许是编写 primeFactorsOnly(n) 函数，让它返回 n 的素因子，但不带乘方。然后将要求归结为检查(a - 1) % primeFactorsOnly(m) == 0。现在来实现 primeFactorsOnly。

有很多方法可用来获得某个数 n 的素因子。有一个简单方法就是素数 $p_1, p_2, p_3, \cdots$，然后反过来检查 p_k 是否能将 n 整除，在这时就让 p_k 乘以一个累加数 r。当 p_k 大于 n 时，累加数就包含了所需要的答案：在 n 的素因子中，每次提取一个。

我知道你们在反问自己：“编译时计算真的有必要吗？”很有必要！请容我慢慢道来。

有一个更简单的版本，就是不管生成素数，而对于从 2 开始不断增加（2, 3, 5, 7, 9, …）的值 k，只是简单地计算 n mod k。只要 k 能整除 n，那么累加器就乘以 k，然后 n 因 k 的所有乘方而“耗尽”，即是说 n 被重新分配成 n/k，只要 k 能整除 n。这样，我们记录除数 k，并且将 n 减小到了 k 不可约的程度。这似乎是一个比较浪费的方法，但请注意，素数的生成可能会是相关工作所需要的，至少在一种简单的实现里是需要的。此想法的实现像下面这个样子：

```
ulong primeFactorsOnly(ulong n) {
   ulong accum = 1;
   ulong iter = 2;
   for (; n >= iter * iter; iter += 2 - (iter == 2)) {
      if (n % iter) continue;
      accum *= iter;
      do n /= iter; while (n % iter == 0);
   }
```

① 不知何故，欧几里得算法总是会很自然地出现在优秀编程书籍里。

自在读书 www.zizidiary.com

```
    return accum * n;
}
```

更新操作 iter + = 2 -(iter == 2)会将 iter 不断增加 2，不过 iter 为 2 时例外——此时，更新之后 iter 为 3。这样 iter 的值包含了 2, 3, 5, 7, 9, …。对任何像 4 那样的偶数进行检查是一种浪费，因为 2 已经被测试过了，而所有为 2 的乘方的数都已从 *n* 里提取出来了。

为什么迭代会在 n > = iter * iter 时继续下去，而不是在 n > = iter 时呢？答案有点微妙。如果 iter 大于 $\sqrt{n}$，并不同于 n 本身，那么可以确定 iter 不会是 n 的除数：如果它是除数，就需要某个乘数 k 使得 n == k * iter；但已尝试过，所有除数都是小于 iter 的，因此 k 必须大于 iter，从而 k * iter 也是大于 n 的，但这是不可能相等的。

现在对 primeFactorsOnly 函数进行单元测试：

```
unittest {
    assert(primeFactorsOnly(100) == 10);
    assert(primeFactorsOnly(11) == 11);
    assert(primeFactorsOnly(7 * 7 * 11 * 11 * 15) == 7 * 11 * 15);
    assert(primeFactorsOnly(129 * 2) == 129 * 2);
}
```

最后，我们还需要一个小封装器，它会针对这 3 个候选的线性同余生成器参数执行 3 个检查：

```
bool properLinearCongruentialParameters(ulong m, ulong a, ulong c) {
    // 边界检查
    if (m == 0 || a == 0 || a >= m || c == 0 || c >= m) return false;
    // c 和 m 互质的
    if (gcd(c, m) != 1) return false;
    // a - 1 可被 m 的所有质因数整除
    if ((a - 1) % primeFactorsOnly(m)) return false;
    // 如果 a - 1 是 4 的倍数，那么 m 也是 4 的倍数。
    if ((a - 1) % 4 == 0 && m % 4) return false;
    // 所有测试通过
    return true;
}
```

下面对 m、a 和 c 的一些常见值进行单元测试：

```
unittest {
    // 坏掉的示例
    assert(!properLinearCongruentialParameters(
        1UL << 32, 210, 123));
    // 《数值分析》[48]
    assert(properLinearCongruentialParameters(
        1UL << 32, 1664525, 1013904223));
    // Borland C/C++ 编译器
    assert(properLinearCongruentialParameters(
        1UL << 32, 22695477, 1));
    // glibc
    assert(properLinearCongruentialParameters(
        1UL << 32, 1103515245, 12345));
    // ANSI C
    assert(properLinearCongruentialParameters(
        1UL << 32, 134775813, 1));
    // Microsoft Visual C/C++
    assert(properLinearCongruentialParameters(
        1UL << 32, 214013, 2531011));
}
```

properLinearCongruentialParameters 看起来工作得很好，因此我们完成了所有线性同余发生器可靠性测试的细节。不要再拖延了，快些坦白吧！所有这些素数操作和因式分解与编译时计算有什么关系呢？相关的东西都在哪里？模板、宏或者怎么叫它都行，它们都在

自在读书 www.zizidiary.com

哪里？是灵活的 static if 吗？还是那令人兴奋不已的代码生成和扩展呢？

好吧，真相就是：你刚刚看到的所有内容都与编译时函数计算有关。给定任意常数 m、a 和 c，你便可以在编译期间计算 properLinearCongruentialParameters，同时不需要对该函数做任何更改或者更改它所调用的函数。D 语言编译器内嵌有一个解释器，它可以在编译期间计算 D 语言的函数——可以使用算术运算、循环、更改、提前返回、数组，甚至超越函数。

所有需要做的就是明确告知编译器：这个计算必须在编译时完成。具体的实现方法有下面几种：

```
unittest {
    enum ulong m = 1UL << 32, a = 1664525, c = 1013904223;
    // 方法 1：使用静态断言
    static assert(properLinearCongruentialParameters(m, a, c));
    // 方法 2：将结果赋给一个枚举量
    enum proper1 = properLinearCongruentialParameters(m, a, c);
    // 方法 3：将结果赋给一个静态值
    static proper2 = properLinearCongruentialParameters(m, a, c);
}
```

我们还没有深入了解过 struct 和 class 的内容，但这里可以展望一下——properLinearCongruentialParameters 的典型用法就是在一个结构或类的内部定义一个线性同余发生器。例如：

```
struct LinearCongruentialEngine(UIntType,
        UIntType a, UIntType c, UIntType m) {
    static assert(properLinearCongruentialParameters(m, a, c),
        "Incorrect instantiation of LinearCongruentialEngine");
    ...
}
```

事实上，上面几行就是从标准模块 std.random 里的同名结构里复制出来的。

将这个测试从运行时移到编译时会带来两个有趣的结果。第一个结果是，LinearCongruentialEngine 可能会将测试延迟到运行时才进行，如通过将它放置到构造函数里面来实现。然而作为一条通用规则，尽早发现错误总是好过最后才发现，尤其是在一个无法控制它是如何被使用的库里更应该这样。静态测试并不能让 LinearCongruentialEngine 的错误实例化发出相应的信号，它会当它们不存在。第二个结果是，使用编译时常量的代码有很好的机会比使用常规变量 m、a 和 c 的代码执行得更快。在现今大多数处理器上，字面常量可以成为指令流的一部分，因此加载它们根本就不会引发额外的内存访问。我们需要面对实现——线性同余引擎并不是最随机的，使用它的主要原因就是速度。

解释过程要比生成代码慢好几个数量级，但相比传统的通过 C++语言模板实现的元编程，这个方法还是要快上许多，且具有更好的可伸缩性。此外，也是情理之中，编译时的活动在某种程度上是“免费的”。

在撰写本书时，这个解释过程仍具有一定的局限性。通常不允许分配类对象和内存（但内建数组可以工作）。不允许静态数据、内联汇编以及不安全的特性（如联合和某些强制类型转换）。在解释期间所能完成操作的限制在持续压力作用下会形成一个封闭层。具体的计划是在 D 语言的安全子集让所有内容在编译期间都可以进行解释。总之，在编译期间对代码进行解释的能力是近年才具有的，并且引发了各种令人兴奋不已的可能性，值得深入探索。

自在读书 www.zizidiary.com

第 6 章

类、面向对象风格

面向对象编程（Object-Oriented Programming，OOP）的发展经历了从一个惹人喜爱的小孩到一个恼人的满脸青春痘的少年，再到今天的一个环境适应力很强的成年人。如今我们不仅对其强大的功能有了更好的理解，同时也对对象技术内在的局限性有了更深的了解。于是编程社区开始关心有利于创建坚实设计的方法，即将 OOP 的优势与其他范型的长处进行结合。这一趋势正变得越来越明显，现今的编程语言要么采取更为折中的特性，要么从设计的一开始就连同其他编程风格一起将 OOP 建立起来。D 语言选择的是后一种方法；至少在某种程度上，它在保持各种不同的编程范型和谐共处方面，已经做得相当出色。本章要探索一下 D 语言的面向对象特性以及它们与语言的其余部分是如何进行集成的。深入学习面向对象的一个很好的起点就是参考 Bertrand Meyer 的经典著作 *Object-Oriented Software Construction*[40]（更正式的疑难解惑，则请参考 Pierce 的著作 *Types and Programming Languages*[46，第 18 章]）。

6.1 类

在 D 语言里封装对象的单位是类（class）。一个类所定义的是一个用于创建对象的“饼干模具”，即定义这些对象的外观。类可以指定常量、每个类所特有的状态、每个对象所特有的状态以及大量的方法。例如：

```
class Widget {
   // 常量
   enum fudgeFactor = 0.2;
   // 共享的不可更改的值
   static immutable defaultName = "A Widget";
   // 分配给每一个 Widget 对象的状态
   string name = defaultName;
   uint width, height;
   // 静态方法
   static double howFudgy() {
      return fudgeFactor;
```

自在读书 www.zizidiary.com

```
    }
    // 方法
    void changeName(string another) {
        name = another;
    }
    // 不可改写的方法
    final void quadrupleSize() {
        width *= 2;
        height *= 2;
    }
}
```

一个类型为 Widget 的对象的创建可以通过 new 表达式来完成（参见 2.3.6.1 节），如 new Widget。通常都是这样调用，并将其结果存储到一个命名的对象里面。为访问 Widget 内部定义的符号，需要在这个符号的前面加上你想要操作的那个对象，以及一个圆点“.”。如果访问的成员是静态的，则使用类名即可。例如：

```
unittest {
    // 访问 Widget 的静态方法
    assert(Widget.howFudgy() == 0.2);
    // 创建一个部件
    auto w = new Widget;
    // 使用该部件
    assert(w.name == w.defaultName); // 或者 Widget.defaultName
    w.changeName("My Widget");
    assert(w.name == "My Widget");
}
```

请注意这里的差异，上面的代码使用的是 w.defaultName，而非 Widget.defaultName。无论在哪里访问静态成员，使用对象名与使用类名的最终效果是一样的。这是因为圆点左边的名称会首先要求进行名字查找，（如有必要）其次才会进行对象识别。不管 w 最终有没有用到，它都会计算。

6.2　对象名就是引用

先进行一个小试验：

```
import std.stdio;

class A {
    int x = 42;
}

unittest {
    auto a1 = new A;
    assert(a1.x == 42);
    auto a2 = a1;
    a2.x = 100;
    assert(a1.x == 100);
}
```

这个实验很成功（所有断言都通过了），它表明 a1 和 a2 本质上是同一个对象：事实上，对 a2 的更改也会返回去更改 a1。它们两个仅仅名字上存在差异，实际就是同一个对象；因此

自在读书 www.zizidiary.com

对 a2 进行更改同样也会影响到 a1。语句 auto a2 = a1;并不会创建额外的类型为 A 的对象，它只是让已有的对象能通过另一个名字了解到。图 6-1 展示了其中的道理。

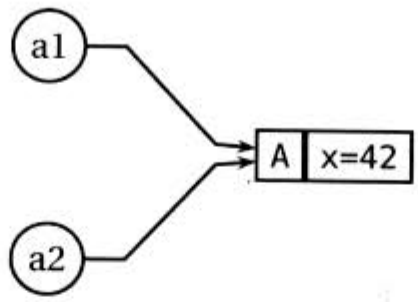


图 6-1　语句 auto a2 = a1;只会针对同一个底层对象增加一个额外的名称

这种行为与“所有类对象都是实体（entity）”的概念是一致的，即它们都具有“个性特征”，如果没有特殊理由请不要进行复制。相比之下，值对象（如内建数字）则具有完整复制的特点。想要定义新的值类型，则可以通过 struct（参见第 7 章）来实现。

因此，在类（class）对象世界里，存在有对象（object），以及对它们的引用（reference）。这种虚构的将引用链接到对象的箭头叫做绑定（binding），例如，我们常说 a1 和 a2 都绑定到同一个对象，或者都拥有相同的绑定。操作一个对象的唯一方式就是使用对它的引用。就对象自身而言，一旦创建，它将永远存在于同一个地方。如果对某个对象感到厌烦了，你所能做的只能是将你的引用绑定到另一个对象上。例如，想要交换两个引用：

```
unittest {
    auto a1 = new A;
    auto a2 = new A;
    a1.x = 100;
    a2.x = 200;
    // 交换 a1 和 a2
    auto t = a1;
    a1 = a2;
    a2 = t;
    assert(a1.x == 200);
    assert(a2.x == 100);
}
```

在标准模块 std.algorithm 里有一个通用程序 swap，我们可以通过调用 swap(a1, a2) 来替换掉上面代码里 assert 前面的 3 条语句。但这里我们并没有这样做，事实上这 3 条语句能够更清晰地展示出它们是如何进行重新连接的。图 6-2 展示了绑定在交换前后的情形。

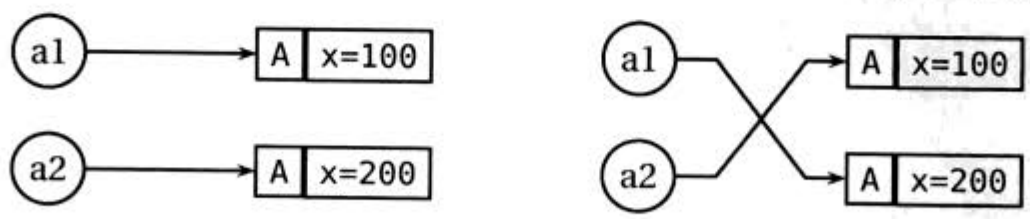


图 6-2　在交换两个引用之前和之后的绑定。整个交换过程改变了引用连线到对象的路径，而这些对象自身仍待在同一个地方

对象自身会留在原地，即在创建后它们在内存中的位置永远不会发生改变。同样值得注意的是，该对象永远不会消失——它一旦创建，你完全可以相信它永远都会在那里。垃圾回收器会在后台回收不再被使用的对象的内存。对象的引用（在这里指的是 a1 和 a2）可以通过重新

自在读书 www.zizidiary.com

绑定指向其他地方。当运行时系统判定出一个对象不再有任何引用绑定到它时，它便会回收这个对象的内存，这种动作称为垃圾回收。这种行为完全不同于值语义（如 int）：在值语义里没有间接性，也没有重新绑定，每一个名字都直接与它所操纵的那个值粘在一起。

未绑定到任何对象的引用为空（null）引用。使用.init 初始化的类引用为 null。引用可以与 null 进行比较，也可以赋值成 null。下面的断言都能顺利通过：

```
unittest {
   A a;
   assert(a is null);
   a = new A;
   assert(a !is null);
   a = null;
   assert(a is null);
   a = A.init;
   assert(a is null);
}
```

访问未绑定的（null）引用会导致硬件故障——它会终止应用程序（或者，在某些系统里和某些条件下，它会启动调试器)。如果你试图访问一个引用的非静态成员，而编译器可以确切地证明该引用一定为 null，那么它将拒绝编译这种代码。

```
A a;
a.x = 5;            // 错误! a 为 null!
```

```
class A { int x; }
A a;
assert(!__traits(compiles, a.x = 5));
```

为了不惹怒大家，编译器是比较保守的——如果某个引用有可能为 null，但不总是一真为 null，那么编译器也会让这种代码顺利通过，并将所有错误推迟到运行时。例如：

```
A a;
if (‹条件›) {
   a = new A;
}
...
if (‹条件›) {
   a.x = 43;        // 正确
}
```

即使在两个计算之间，‹条件›有可能会把值更改掉，编译器还是会让这种代码顺利通过。在一般情况下，人们很难对特定的对象初始化进行验证，因此编译器会假定你清楚自己在做什么。不过最简单的情形除外，即编译器可以确定你是在试图以一种错误的方式使用一个空引用的情形。

D 语言处理类对象的引用语义方法在许多面向对象语言里都能找到相似之处。对类对象使用引用语义和垃圾回收，有积极的和消极的两方面的结果（分别用+和–标识），具体如下。

+多态性。持续使用引用所带来的间接性的层次为多态性提供了支持。所有引用都具有同样的大小，但与之关联的对象却可以拥有不同的大小，即使它们表面上拥有相同的类型（通过使用继承，继承很快会讨论到）。因为引用拥有相同的大小，且与它们所引用的对象大小无关，因此总可以将对派生对象的引用替换成对基对象的引用。即使数组里的实际对象可能拥有不同的大小，对象数组同样可以正常工作。如果用过 C++，那么你肯定知道将指针与多态性一起使用的必要性，并且也知道当忘记这一点时将会遭遇的各种致命的问题。

自在读书 www.zizidiary.com

+安全性。有许多人都认为垃圾回收仅仅是带来了方便，即它通过减轻程序员对内存管理的负担简化了编码。不过也许有些令人惊讶，在无限生命期模型（垃圾回收让它得以实现）和内存安全之间却有着很强的联系。因为有了无限生命期，才不会出现悬空引用，即所引用的对象已不存在以及所引用对象的内存已被另一个不相关的对象重新占用。请注意，完全使用值语义（执行 `auto a2 = a1;`，则会将 a1 所引用的 A 类型对象进行复制，而让 a2 引用到该对象的副本）则是安全的。然而，这种方案几乎没什么意义，因为它无法创建任何引用结构（如列表、树、图，以及更为通用的共享资源）。

-分配成本。通常情况下，类对象必须要驻留在垃圾回收堆里，与在栈上进行分配相比，这种方式普遍要慢些，并且会消耗更多的内存。近年来这种差距已大大缩小，但仍然存在。

-远程耦合。使用引用的主要风险是过度别名化。完全使用引用语义很容易出现这样的情况：多个指向同一个对象的引用驻留在不同的（完全想不到的）地方。在图 6-1 里，a1 和 a2 之间的距离，就应用程序逻辑而言，可以是任意的，并且还有可能存在许多的其他引用也挂接在这一个对象上。有意思的是，如果被引用的对象是不可变的（immutable），那就不会有什么问题——只要没人修改这个对象，就不会出现耦合。在应用程序的里，当某个特定环境里产生的变化会出人意料地影响到其另一部分正监视的状态时，麻烦就会出现。进行显式复制是解决此问题的一种方法，即只要对象进行传递，就调用一个特殊的名叫 `clone`（克隆）的方法。这种方法的缺点是：它是基于准则的，并且如果应用程序的好几个部分都觉得最好是克隆对象才不会出现问题，则可能会导致效率低下。

对比一下引用语义与值语义（如 `int`）。值语义是有优势的，尤其是在方程式推理方面：在表达式里，等式总是可以进行互换，同时结果不会改变。相比之下，引用是使用方法调用来修改底层对象的，因此不支持这样的推理。速度也是值语义的一个重要优势，但是如果你想要拥有多态性的动态行为，那么引用语义是必需的。与将所有类型都统一成引用语义的纯面向对象语言相比，有些语言也试过将两者整合在一起，它们最终得到了“混杂”（impure）的绰号。D 语言是混杂的，并且对此毫不掩饰。针对特定的类型，在设计时你可以选择是否要使用 OOP：选择 OOP，就需要使用类；否则，如果使用结构，便会失去 OOP 特有的与引用语义密不可分的便利。

6.3 对象的生命周期

鉴于我们对对象的去向已有了一个通用的概念，现在来详细了解一下它们的生命周期（life cycle）。创建对象时，可以使用 `new` 表达式：

```
import std.math;

class Test {
    double a = 0.4;
    double b;
}

unittest {
    // 使用 new 表达式来创建对象
    auto t = new Test;
```

自在读书 www.zizidiary.com

```
    assert(t.a == 0.4 && isNaN(t.b));
}
```

执行 new Test 会构造出一个默认初始化的 Test 对象，其类型为 Test，其中每一个字段都被初始化成各自的默认值。每种类型 T 都有一个静态已知的默认值，它可以通过 T.init（关于各种基本类型的.init 值，参见表 2-1）进行访问。如果想将某些成员变量初始化成与其类型的.init 值不同的内容，则可以在定义它时，指定一个静态已知的初始值，如上面示例里的变量 a。上面的单元测试之所以能够通过，是因为 a 显式初始化成了 0.4，并且因为 b 未做任何更改，所以它会隐式初始化成 double.init，即“不是一个数字”（NaN）。

6.3.1　构造函数

当然，有的时候使用某些静态已知的值来初始化字段是远远不够的。为了在创建对象时执行代码，可以使用称为构造函数（constructor）的特殊函数。构造函数指的是一个名称为 this 且没有返回类型的函数：

```
class Test {
    double a = 0.4;
    int b;
    this(int b) {
        this.b = b;
    }
}
unittest {
    auto t = new Test(5);
}
```

只要一个类至少定义了一个构造函数，那么其隐式的默认构造函数便不再可用。对于上面定义的 Test，下面的测试将不再工作：

```
auto t = new Test;
```

定义这个禁止操作的目的在于尽量避免一个常见的错误：设计师在精心定义一些带参数的构造函数时，完全忘记了默认构造函数这回事。在 D 语言里，这也是很常见的情况。这种对“健忘症”的保护也很容易取消，只需简单地告诉编译器“是的，我还记得”即可：

```
class Test {
    double a = 0.4;
    int b;
    this(int b) {
        this.b = b;
    }
    this() {} // 默认构造函数，所有字段都会隐式初始化
}
```

在方法内部（静态方法除外，参见 6.5 节），this 引用会被隐式绑定到接收此调用的对象上。有时这个引用是很有用的，正如上面的示例，它演示了一种常见的在构造函数里的命名习惯：如果某个参数是要初始化一个成员，而它们的名字是相同的，那么在成员名字的前缀显式加上一个引用 this 便可以消除成员与参数之间的歧义。如果没有前缀，则参数会掩盖掉与它同名的成员。

虽然可以通过 this.field 方式来修改任何字段，但绝不能重新绑定 this 自身，这样做

自在读书 www.zizidiary.com

它就会表现得像一个右值：

```
class NoGo {
    void fun() {
        // 将 this 重新绑定到一个不同的对象上
        this = new NoGo; // 出错！不能修改"this"！
    }
}
```

平常的函数重载规则（参见5.5节）也可以用于构造函数：一个类可以定义任意数量的构造函数，前提是它们要有不同的签名（不同的参数个数或者至少有一个参数的类型不相同）。

6.3.2　转发构造函数

假设有一个类 Widget，它定义了两个构造函数：

```
class Widget {
    this(uint height) {
        this.width = 1;
        this.height = height;
    }
    this(uint width, uint height) {
        this.width = width;
        this.height = height;
    }
    uint width, height;
    ...
}
```

上面的代码存在不少重复的地方，并且对于更大型的类情况只会变得更糟，不过幸运的是，一个构造函数可以交由另一个构造函数来处理：

```
class Widget {
    this(uint height) {
        this(1, height); // 转移到另一个构造函数
    }
    this(uint width, uint height) {
        this.width = width;
        this.height = height;
    }
    uint width, height;
    ...
}
```

在显式调用构造函数时，如 this(1, h)，会有一些限制。首先，只能从另一个构造函数内部发出这样的调用。其次，如果你真的决定要进行这样的调用，那么你必须“说服”编译器：不管什么情况，在整个构造函数里你都恰好执行了一次这样的调用。例如，下面的构造函数是无效的。

```
this(uint h) {
    if (h > 1) {
        this(1, h);
    }
    // 出错！有一条路径跳过了构造函数
}
```

在上面的示例里，编译器会认为存在不会调用另一个构造函数的情况，并且将这种情况当成一个错误。这样做的目的在于想让构造函数要么构造出自己，要么将构造任务转移给另一个构造函数。

自在读书 www.zizidiary.com

构造函数可能选择转移至另一个构造函数，也可能选择不转移，对于这样的情况是不允许的。

两次调用同一个构造函数也是错误的：

```
this(uint h) {
    if (h > 1) {
        this(1, h);
    }
    this(0, h);
}
```

与忘记初始化相比，大家更不会想要一个被双重初始化的对象，因此这种情况会被标记为一个编译时错误。简而言之，一个构造函数可以调用另一个构造函数，调用次数要么完全为零要么恰好只有一次。在编译期间，这一要求会通过使用简单流分析的方式来进行验证。

6.3.3　构造步骤

在所有语言里，对象的构造都有点棘手。构造过程起始于一块无类型的内存，接着存入信息，让这块内存看起来像一个类对象。其中还是需要一些“魔法”的。

在 D 语言里，对象的构造包含了以下几个步骤。

（1）分配。运行时会在堆上分配一块原始内存，其大小足以存储对象的所有非静态字段。所有基于类的对象都是动态分配的——与 C++有所不同，D 语言不支持在栈上分配类对象。如果分配失败，构造过程便会停止，并抛出一个异常。

（2）字段初始化。每个字段都被初始化成它的默认值。如上面所讨论的，默认的字段值可以是在字段声明里使用“= 值”的方式进行指定；如果没有使用这种方式，则使用该字段所属类型的.init 值。

（3）冠名。在默认字段初始化之后，该对象会被标明为完整的 T 类型，这个操作甚至是在实际的构造函数还未被调用之前完成的。冠名过程是由具体的编译器实现决定的，通常包含的是使用与类型相关的信息对一些隐藏字段进行初始化操作。

（4）构造函数调用。最后，编译器引发一个对可用构造函数的调用。如果该类没有定义构造函数，则跳过这一步。

因为对象在默认字段初始化完成之后便被认为是存活着的，因此强烈建议字段的初始化值应总是将对象置于一个有意义的状态。然后，再由实际的构造函数进行调整，从而将对象置于一个有意义的状态（当然，这也是很有意义的）。

如果构造函数对某些字段进行了重新赋值，那么这种双重赋值不会存在效率问题。多数时候，如果构造函数的函数体很简单，那么编译器会发现第一次赋值是多余的，于是进行黑色幽默式的“消除无效赋值”优化。

如果真的很在意效率问题，那么也可以将= void 指定为所在意字段的初始值。这时，在构造函数里初始化该成员时就要非常小心。对于固定大小的数组，使用= void 特别有用。让编译器来优化数组元素的双重初始化非常困难，所以需要人为地帮助它。下面的代码使用“0.0, 0.1,0.2,…,12.7”很高效地初始化了一个固定大小的数组：

自在读书 www.zizidiary.com

```
class Transmogrifier {
    double[128] alpha = void;
    this() {
        foreach (i, ref e; alpha) {
            e = i * 0.1;
        }
    }
    ...
}
```

有时，有的设计可能会故意要求不初始化某些字段。例如，Transmogrifier 通过一个单独的变量 usedAlpha（开始时为零）便可以跟踪到 alpha 已经使用了的那部分。对象的原语知道 alpha 里只有部分（从 0 到 usedAlpha 这段范围）实际上是被初始化的：

```
class Transmogrifier {
    double[128] alpha = void;
    size_t usedAlpha;
    this() {
        // 让 usedAlpha = 0 而 alpha 未初始化
    }
    ...
}
```

开始时 usedAlpha 为零，这是 Transmogrifier 需要执行的所有初始化操作。随着 usedAlpha 的增大，这段代码一定不会在写入之前读取 alpha 里处于区间[usedAlpha .. $]内的元素。当然，这是你（而非编译器）需要确保的事情——它充分说明有时存在效率和静态可验证性之间不可避免的牵制关系。尽管这种优化常常是微不足道的，但在某些情况下，这种有些多余的强迫性初始化可能会显著地影响到最终结果，所以有一个“安全出口”可供选择总会令人安心。

6.3.4　析构与解除分配

D 语言为所有类对象维护着一个垃圾回收堆。一旦某个对象被成功分配，那么相对应用程序而言，它便永远存活着。只要垃圾回收器能够证明对某个对象的可访问引用不再存在，它便会回收这个对象使用过的内存。这种方案让基于类编写的代码变得更加整洁、安全。

对于某些类，在终止进程上设置一个“钩子”是很重要的，这样它们便能够释放掉它们可能已请求过的附加资源。这些类可以定义析构函数（destructor），这是一个比较特殊的函数，其名字为~this：

```
import core.stdc.stdlib;

class Buffer {
    private void* data;
    // 构造函数
    this() {
        data = malloc(1024);
    }
    // 析构函数
    ~this() {
        free(data);
    }
```

自在读书
www.zizidiary.com

```
    ...
}
```

上面的示例演示了一种极端情况——这是一个自己维护着原始内存缓冲区的类。多数时候类使用的是完全封装好的资源，所以没必要定义析构函数。

6.3.5　拆毁步骤

与构造操作一样，拆毁一个对象也要遵循一些规定。

（1）对象在冠名（构造过程的第 3 步骤）之后，立即便被认为是活的，并处于垃圾回收器的监视之下。请注意，即使自定义构造函数后来会抛出异常，这种情况也还是一样的。就垃圾回收器而言，考虑到默认初始化和冠名不能失败，因此也就意味着一个成功分配的对象会被认为是完整的对象。

（2）对象在整个程序里都可以使用。

（3）对象的所有可访问引用都会消失，不会再有代码能到达该对象。

（4）视具体实现情况而定，系统认可对象的内存可以回收，并会调用其析构函数。

（5）在稍后的某个时间点上（要么恰在调用析构函数之后，要么再晚一点）系统会重新使用该对象的内存。

关于最后两个步骤有一个细节很重要，即垃圾回收器会确保对象的析构函数不能访问一个已被解除分配的（deallocated）对象；但可以访问一个已被销毁的（destroyed）对象，因为它还未被解除分配。在 D 语言里，已被销毁的对象还会暂时继续占用内存，直到它们的引用也被拆毁为止。如果不这样做，那么在执行销毁和解除分配以循环方式相互引用的（如循环列表）对象时，这些操作便无法安全实现。

上面描述的生命周期也存在多种修订方法。第一种修订方法是，应用程序在执行到步骤 4 之前很有可能结束。通常小型的应用程序都运行在内存很充足的系统里。在这种情况下，D 语言会假定退出应用程序实际上会释放掉所有与之关联的资源，因此它不会调用任何析构函数。

另一种修订对象生命周期的方法是显式地调用对象的析构函数。这一点可以通过调用 object 模块（在任何编译过程中会被自动导入的标准库模块）里定义的 clear 函数。

```
unittest {
    auto b = new Buffer;
    ...
    clear(b); // 去年 b 的附加状态
    ... // b 在这里仍然可用
}
```

通过调用 clear(b)，用户所表达的意图是：显式调用 b 的析构函数（如果存在）、使用 Buffer.init 来抹除对象的状态，并且调用 Buffer 的默认构造函数。不过与 C++有所不同，clear 并不会释放掉（dispose）对象自己的内存，并且也不能使用 delete 操作符。D 语言有一个 delete 操作符，但它已被弃用。你仍然可以手动释放内存，前提是你真的清楚自己在干什么，具体方法就是调用在模块 core.memory 里可以找到的函数 GC.free()。释放内存是

自在读书 www.zizidiary.com

不安全的，但调用 clear 是安全的，因为内存实际上并没有消失，因此也就不会存在悬空指针的风险。在执行 clear(obj)之后，即使对象 obj 不包含任何有意义的状态，但不管是出于什么目的的访问和应用，它都没有问题。例如，下面的 D 语言代码是正确的：

```
unittest {
    auto b = new Buffer;
    auto b1 = b;                    // b 的附加别名
    clear(b);
    assert(b1.data !is null);       // 附加别名仍然引用 b 的（有效）底层
}
```

因此在调用 clear 之后，该对象仍然存在，但其析构函数已被调用，因此该对象现在携带的是其默认构造时的状态。有意思的是，在下一个垃圾回收期间，此对象的析构函数会被再次调用。很显然，这是因为垃圾回收器并不知道你已将这个对象置于什么状态。

为什么会出现这种行为呢？答案很简单——将对象的内存解除分配与拆毁操作区分开来，让你既可以手动控制对象可能仍在控制的昂贵资源（如文件、套接字、互斥锁和系统处理），同时又能确保内存的安全性。使用 new 和 clear 绝不会创建出悬空指针。当然，如果你手“发痒”，使用了 C 语言的 malloc 和 free 方法，或者上述的 GC.free，那么你便会遇到悬空指针的危险。通常情况下，明智的做法是将资源处理（安全的）与内存回收（不安全的）分隔开来。内存完全不同于其他任何资源，因为它是类型系统的物理支撑。如果不经意将它解除分配的话，则可能破坏类型系统所承诺的任何保证。

6.3.6　静态的构造函数与析构函数

在类的内部，本质上说是 D 语言的任何地方，静态数据都必须要使用编译时常量进行初始化。为了让线程启动期间代码能有序地执行，编译器允许定义特殊函数 static this()。执行模块级和类级初始化操作的代码被集中到一起，这样运行时支持库便能以有序的方式进行静态初始化。

在类的内部，可以定义多个的带有 static 前缀的默认构造函数：

```
class A {
    static A a1;
    static this() {
        a1 = new A;
    }
    ...
    static this() {
        a2 = new A;
    }
    static A a2;
}
```

这种函数称为“静态类构造函数”。当加载应用程序时，并且在执行 main 函数之前，编译器会依次执行每一个静态类构造函数，其顺序与它们在源代码里的出现的顺序一致。在上面的示例里，a1 将在 a2 之前被初始化。在同一模块不同类里的静态类构造函数，其执行顺序与词法顺序一致。在不相关模块里的静态类构造函数，其执行顺序不定。最后还有一点，在相互依

自在读书 www.zizidiary.com

赖的模块里的各个类的静态类构造函数，其排序的目的则是为了消除这样一种可能：在运行某一个类的 static this()之前便已用到它。

顺序是如何发挥作用的呢？假设类 A 定义在模块 MA 里，而类 B 定义在模块 MB 里，则可能发生以下情况。

- A 和 B 至多只有一个定义了一个静态类构造函数。这时，可以不用考虑顺序的问题。
- MA 和 MB 都未彼此导入。这时，执行顺序不定——任何顺序都可以工作，因为两个模块彼此并不依赖。
- MA 导入 MB。这时，B 的静态类构造函数会在 A 之前运行。
- MB 导入 MA。这时，A 的静态类构造函数会在 B 之前运行。
- MA 导入 MB，同时 MB 导入 MA。这时，会产生一个“循环依赖”（cyclic dependency）的错误信息，并且程序加载过程中的执行会中止。

这里的推理过程并不完全取决于类 A 和类 B，而只是取决于模块本身以及它们的导入关系。在第 11 章会对这一点进行详细讨论。

如果某个静态类构造函数因抛出异常而失败，那么程序会立即终止。如果全部成功，则类便有机会通过定义静态类析构函数，在线程关闭期间进行一些清理工作。预计的效果看起来有点像下面这个样子：

```
class A {
   static A a1;
   static ~this() {
      clear(a1);
   }
   ...
   static A a2;
   static ~this() {
      clear(a2);
   }
}
```

静态析构函数会在线程关闭期间运行。在各个模块里，它们运行的顺序与定义顺序相反。在上面的示例里，a2 的析构函数会在 a1 的析构函数之前被调用。当涉及多个模块时，模块调用静态类析构函数的顺序恰好与模块调用其静态类构造函数的顺序相反。它们的相反情况与之类似。

6.4　方法和继承

我们现在已对对象的创建和清除有所了解，下面来看看如何使用它们。与对象的大部分交互都是通过调用其方法实现的。在某些语言里，这样的动作也称为给对象“发送消息”。方法的定义看起来与常规函数的定义没什么两样，唯一的区别在于它是出现在类里。为将描述集中到示例上，现假设要构建一个 Rolodex 应用程序，它允许你存储和显示联系人信息。于是，这些信息（一张虚拟商务卡）的单位便是 Contact 类。另外，它可能还会定义某个方法，用于提供显示联系人时所使用的背景颜色：

```
class Contact {
```

自在读书 www.zizidiary.com

```
    string bgColor() {
        return "Gray";
    }
}
unittest {
    auto c = new Contact;
    assert(c.bgColor() == "Gray");
}
```

当你觉得一个类应该继承另一个类时，事情便开始变得有意思了。例如，某些联系人是朋友，我们需要为他们定制出不同的背景色：

```
class Friend : Contact {
    string currentBgColor = "LightGreen";
    string currentReminder;
    override string bgColor() {
        return currentBgColor;
    }
    string reminder() {
    ...
        return currentReminder;
    }
}
```

通过记号: Contact 可以声明 Contact 的继承，类 Friend 的实例会包含 Contact 的全部内容以及 Friend 附带的状态（本例中的 currentBgColor 和 currentReminder）和方法（本例中的 reminder）。

我们将 Friend 称作 Contact 的子类，而将 Contact 称作 Friend 的父类。子类的好处在于，不管在什么地方，当期望的是 Contact 值时，你都可以将其替换成 Friend 值：

```
unittest {
    Friend f = new Friend;
    Contact c = f;                  // 使用 Friend 代替 Contact
    auto color = c.bgColor();       // 调用 Friend 的方法
}
```

如果替换后的 Friend 在行为上恰好跟 Contact 一样，那么就没什么必要使用 Friend 了。对象技术的其中一个关键特点就是它允许派生类改写基类中的函数，这样便能以一种模块化的方式对行为进行定制。毫无疑问，关键字 override（在 Friend 的 bgColor 定义中）即表示进行改写，也就是说，对于一个实际上已被替换成 Friend 类型的 Contact 对象，调用 c.bgColor() 则总是会调用 Friend 里的方法版本。Friend 永远都是 Friend，即使编译器把它们认作是简化的 Contact 时也一样。

6.4.1　术语大杂烩

对象技术在学术界和工业界的发展已经历了一段悠长而成功的历史。目前，它已经成为许多工作的重点，并催生了大量的术语，而这些术语常常会引起混淆。让我们稍停片刻，回顾一下这些术语。

如果类 D 直接继承了类 B，那么类 D 就称为类 B 的子类（subclass）、类 B 的孩子类（child

自在读书 www.zizidiary.com

class）或者从类 B 派生的（derived）类。相应地，类 B 称为类 D 的超类（super class）、父类（parent class）或基类（base class）。

只有当类 X 是类 B 的孩子，或者类 X 是类 B 的孩子的后代时，类 X 才就是类 B 的后代。这是一个递归定义，换句话说，如果你沿着 X 的父级，接着是 X 的父级的父级以此向上走，你总会在某个点遇到 B。

本书通篇使用的是“父/子”（parent/child）和“祖先/后代”（ancestor/ descendant）的概念，因为这些短语比“超类/子类”（superclass/subclass）能更准确地区分“直接继承”（direct）与可能有的“间接继承”（indirect）这样一些概念。

令人奇怪的是，尽管类是类型，但子类型（subtype）与子类不是一回事儿；同样地，超类型（supertype）与超类也不是一回事儿。子类型是一个更通用的概念，也就是说，如果类型 S 的值可以安全地在期望类型为 T 的上下文里使用时，类型 S 是类型 T 的一个子类型。请注意，这个定义并未要求和提及继承。事实上，继承只是实现子类型的一种方法，通常（在 D 语言里）还有其他的方法。子类型与继承之间的关系就是类 C 的后代加上 C 自身都是 C 的子类型。C 的子类型中与 C 不同的子类型才是一个“真正的子类型”。

6.4.2　继承就是子类型化，静态类型和动态类型

我们现在来举例说明继承是如何引发子类型化（subtyping）的。如前面所述，一个派生类对象总是能替换基类对象：

```
class Contact {...}
class Friend : Contact {...}
void fun(Contact c) {...}
unittest {
   auto c = new Contact;          // c 的类型为 Contact
   fun(c);
   auto f = new Friend;           // f 的类型为 Friend
   fun(f);
}
```

尽管 fun 期望的是 Contact 对象，但传递 f 也能达到同样的效果，因为 Friend 是 Contact 的一个子类（同时也是一个子类型）。

当子类型化起作用时，常存在这样的可能，即对象实际的类型会被部分“遗忘”。假设：

```
class Contact { string bgColor() { return ""; } }
class Friend : Contact {
   override string bgColor() { return "LightGreen"; }
}
```

```
unittest {
   Contact c = new Friend;        // c 的类型为 Contact,
                                  // 但实际引用的是 Friend
   assert(c.bgColor() == "LightGreen");
                                  // 它其实是一个朋友!
}
```

假设 c 的类型为 Contact，那么它只能使用 Contact 限定的方法，即使它绑定的是一

自在读书 www.zizidiary.com

个类型为 Friend 的对象也一样。例如，你不能调用 c.reminder，因为该方法仅限于 Friend，在 Contact 里它并不存在。然而，上面的断言（assert）显示朋友永远都是朋友：调用 c.bgColor 表明实际调用了特定于 Friend 的方法。正如在 6.3 节中所讨论的，一旦对象被构造，它便会永远存在，因此使用 new 创建的 Friend 对象永远不会消失。这里有一段小插曲，即绑定到它的引用 c 所拥有的类型为 Contact，而不是 Friend。在这种情况下，我们就说 c 拥有静态类型 Contact 和动态类型 Friend。未绑定的（null）引用没有动态类型。

这里很详尽地对隐藏于 Contact 伪装下面的 Friend 进行了梳理（一般情况下是从祖先到后代）。一方面，这种操作可能失败：如果这里的 c 实际上并不是真的引用到了 Friend，那么结果会怎么样呢？多数时候，编译器无法辨别出这种情况。做这种提取需要依靠强制类型转换 cast：

```
unittest {
    auto c = new Contact;          // c 拥有静态和动态类型 Contact
    auto f = cast(Friend) c;
    assert(f is null);             // f 拥有静态类型 Friend，并且未被绑定
    c = new Friend;                // 静态类型：Contact，动态类型：Friend
    f = cast(Friend) c;            // 静态类型：Friend，动态类型：Friend
    assert(f !is null);            // 通过！
}
```

6.4.3　改写只能是自愿的

在 Friend.bgColor 的签名里，关键字 override 是必需的，这点乍看起来有点烦人。编译器知道改写就是完全让自己有效，并恰当地将各部分内容连接起来。既然如此，那为什么还说 override 是必需的呢？

答案与可维护性有关。事实上，让编译器自动弄清楚你想要改写的是哪些方法并不是件麻烦的事情。问题在于，没有办法确定哪些方法是你不想要改写的。当你定义派生类之后又想要更改基类时，便可能出现这种情况。想象一下，假如 Contact 一开始只定义了 bgColor 方法，再由它派生出 Friend，并改写 bgColor，结果同上面代码片段所示。在 Friend 里，你可能还定义了另一个方法，如 Friend.reminder，它允许你查询与某个特定朋友相关的提醒。如果后来其他人（包括 3 个月后的你）又为 Contact 定义了一个具有不同意义的 reminder 方法，那么你现在便拥有了一个奇怪的错漏，即当对 Contact.reminder 的调用传递到 Contact 或 Friend 时，它实际上执行的是 Friend.reminder，而这是 Friend 未曾预料到的。

相反的情况同样有害，只是没那么严重。比方说，在完成最初设计之后，Contact 决定删除某个方法或更改它的名字。此时，设计者便不得不自己动手遍历所有 Contact 的派生类，以决定如何处理那些现已孤立的方法。当基类设计者对层次结构的某些部分也不可写时，这便是一种错误频发的举动，并且有时还无法在其实体中开展。

自在读书
www.zizidiary.com

所以强制 override 能让你在修改基类的同时，不会对派生类产生意想不到的危害。

6.4.4　调用被改写的方法

有时，改写方法想要调用的恰巧是它正改写的那个方法。假如，有一个图形部件 Button，它继承了一个 Clickable 类。这个 Clickable 类知道如何将按钮的按动分发给监听者，但它对视觉效果并不关心。为引入视觉反馈，Button 改写了 Clickable 定义的 onClick 方法，并在引入视觉效果的同时还想要调用 Clickable.onClick 以执行分发功能。

```
class Clickable {
   void onClick() {...}
}
class Button : Clickable {
   void drawClicked() {...}
   override void onClick() {
      drawClicked();        // 引入图形效果
      super.onClick();      // 向侦听者派发点击动作
   }
}
```

要调用被改写的方法，请使用预定义的别名 super，它会指示编译器去访问方法，而该方法定义在父类里。你可以随意调用任何方法，并不仅限于当前正被改写的方法，如可以从 Button.onClick 内部产生对 super.onDoubleClick() 的调用。老实说，被访问的符号甚至不一定是方法名，还可以是字段名，或者任何符号。例如：

```
class Base {
   double number = 5.5;
}
class Derived : Base {
   int number = 10;
   double fun() {
      return number + super.number;
   }
}
```

Derived.fun 访问了自己的成员和基类中的成员，它们有时具有不同的类型。

一般情况下，访问祖先里的成员的方法是使用“类名.成员名”。事实上，super 只不过是当前基类所具有的任意名字的别名。在上面的示例里，Base.number 的写法完全等同于 super.number。最显著的差别在于，super 使代码具有更好的可维护性：如果你想改变类的基，则完全不需要去查找和替换这些名称。

使用明确的类名，则能跳过多个继承层次。使用 super 或类名显式地修饰方法名称也会更快一些，因为编译器可以准确知道该调度哪一个函数。如果这里涉及的符号不是一个可改写的方法，那么显式的修饰就只会影响到可见性，而不会影响速度。

尽管析构函数（参见 6.3.4 节）也是方法，但它的调用处理却有些不同。你不能发起对父级析构函数的调用，但是当调用析构函数时（无论是在垃圾回收期间还是在对 clear(obj) 请求的响应里），D 语言的运行时支持总是沿着层次结构向上调用所有的析构函数。

自在读书 www.zizidiary.com

6.4.5 协变返回类型

继续 Widget、TextWidget 和 VisibleTextWidget 这几个示例，假设要添加代码，让它复制一个 Widget。在这种情况下，如果被复制的对象为 Widget，那么该副本也将是一个 Widget；如果它是一个 TextWidget，则副本也将是一个 TextWidget；以此类推。一种实现正确复制的方法是在基类里定义一个 duplicate 方法，并要求每个派生类都实现它：

```
class Widget {
    ...
    this(Widget source) {
        ... // 复制状态
    }
    Widget duplicate() {
        return new Widget(this); // 分配内存，并调用 this(Widget)
    }
}
```

到目前为止，一切都很顺利。现在来看看 TextWidget 类里的改写操作：

```
class TextWidget : Widget {
    ...
    this(TextWidget source) {
        super(source);
        ... // 复制状态
    }
    override Widget duplicate() {
        return new TextWidget(this);
    }
}
```

全部都正确，但静态信息明显有丢失：TextWidget.duplicate 实际上返回的是一个 Widget 对象，而不是 TextWidget 对象。但只要我们深入函数查看一下，就能知道 TextWidget.duplicate 的结果类型是 TextWidget。然而，这些信息随着 TextWidget.duplicate 的返回被丢失了，因为 TextWidget.duplicate 的返回类型是 Widget——与 Widget.duplicate 的返回类型一样。因此，尽管下面的代码在一个完美世界里应该工作，但实际上它并不工作：

```
void workWith(TextWidget tw) {
    TextWidget clone = tw.duplicate(); // 出错!
        // 不能将 Widget 转换成 TextWidget!
    ...
}
```

为尽量多地提供静态类型信息，D 语言定义了一种称作“协变返回类型”（covariant return type）的特性。听起来有些时髦，返回类型的协变其实相当简单：如果一个基类型返回的是某个类型为 C 的类，那么改写函数可以返回的类型不仅可以为 C，也可以是任何派生自 C 的类。利用这种特性，你便可以将 TextWidget 声明为 TextWidget.duplicate 的返回类型。还有重要的一点，在交谈中偷偷夹杂一点“协变返回类型”可以让你听起来相当厉害。（开玩笑的。真的！千万别这样做。）

自在读书 www.zizidiary.com

6.5　静态成员的类级封装

有时不仅封装字段和方法有用，将常规函数和（喘口气）全局数据封装进一个类也是很有用的。这种函数和数据除了作用域仅限于类的内部以外，也没什么特别之处。为了在类的所有对象之间共享一些固定的函数和数据，可以使用关键字 static 来引入它们：

```
class Widget {
   static Color defaultBgColor;
   static Color combineBackgrounds(Widget bottom, Widget top) {
      ...
   }
}
```

在静态方法内部，没有 this 引用。同样地，这是因为静态方法就是常规的作用域仅限于类内部的函数。在逻辑上，不需要通过对象来访问 defaultBgColor 或调用 combineBackgrounds——只需在其前缀加上它们的类名即可：

```
unittest {
   auto w1 = new Widget, w2 = new Widget;
   auto c = Widget.defaultBgColor;
   // 这个也有效：w1.defaultBgColor;
   c = Widget.combineBackgrounds(w1, w2);
   // 这个也有效：w2.combineBackgrounds(w1, w2);
}
```

访问静态成员时，使用对象来代替类名，也是没问题的。请注意，即使不是真正需要对象值，它也会被计算：

```
// 创建一个 Widget，并抛弃它
auto c = (new Widget).defaultBgColor;
```

6.6　使用 final 方法来限制扩展性

有时会很想禁止子类改写某个方法。这种情况很常见，因为有些方法并不适合当做定制点。这类方法也可以调用可定制方法，但更常见的情况是：你并不想要更改某些控制流。（突然间想到了模板方法设计模式[27]。）为防止继承类改写某个方法，需要在其定义前面加上 final。

例如，假设有一个股票报价器应用程序，它要确保不管什么时候股票价格改变，便立即更新屏幕上显示的信息：

```
class StockTicker {
   final void updatePrice(double last) {
      doUpdatePrice(last);
      refreshDisplay();
   }
   void doUpdatePrice(double last) {...}
   void refreshDisplay() {...}
}
```

方法 doUpdatePrice 和 refreshDisplay 都是可改写的，因此它们为子类 StockTicker 提供了定制点。例如，有些股票报价器会在价格出现某些变化时产生触发和报

自在读书 www.zizidiary.com

警事件，或者自己以特定的颜色进行显示。但是 doUpdatePrice 是不能被改写的，因此调用者可以确定只要股票价格不更新，显示也不会随之更新。事实上，如果只是为了保证正确性，那么我们可以将 updatePrice 定义成下面这个样子：

```
final void updatePrice(double last) {
    scope(exit) refreshDisplay();
    doUpdatePrice(last);
}
```

借助于 scope(exit)，这里的显示会被正确地刷新，即使 doUpdatePrice 会抛出异常也一样。这种方法真正地保证了显示所反映的是对象最新、最好的状态。

还需要了解一点额外的与 final 方法相关的内容：这种方法是有危险的，因为它很容易将你引向过早优化的阴暗面。其真相就是，final 方法可能更具有效率。因为非 final 方法在每次调用时都会有一个间接步骤，这个步骤保证了 override 所带来的灵活性。对于某些 final 方法，间接性仍然是必要的。例如，当通过基类对象进行调用时，方法的最终改写通常仍会受制于间接调用。在一般情况下，编译器仍然不知道调用该从哪里开始。但如果 final 方法同时也是一个首次引入的方法（而非由基类方法改写而来的），那么当调用它时，编译器便可以 100%确定该从哪里开始调用。所以，final 非改写方法绝不会受制于间接调用；事实上，它们具有与普通函数一样的特点，如调用协定、低开销以及内联机会。表面上看起来，final 非改写方法会比其他方法快很多，但这种优势却因以下两种因素大打折扣。

第一个因素是，调用开销基线是利用一个什么也不做的函数来进行评定的。为评定关键的开销，除调用开销外，函数内部实际花费的时间也必须包含在内。如果函数很短，那么相对开销则可能很大；但如果函数做了一些重要的工作，相对开销则会迅速减少，直到变成噪声。第二个因素是，各种编译器、链接器和运行时优化技术都在积极寻求办法减少或者消除调度开销。你最好能从一开始就审慎地考虑灵活性和进行优化，不要一开始就为了某种未来潜在的性能问题，采用会使其变得过于刚硬的方式来削弱你的设计。

如果你使用的是 Java，那么应能立即认出 final 来，因为在 D 语言里，它具有与之相同的语义。如果将这一点与 C++进行比较，你会注意到一个有趣的在默认情形下的差异：在 C++里，如果不使用任何说明，那么这些方法在默认情况下是最终的（final）；如果显式地将它们说明为虚的（virtual），那么这种方法便是非最终的。同样地，至少在这种情况里，默认支持灵活性会更好一些。通常只会在支持设计决策时，才会想到使用 final，并且很少会将它当做消除额外周期的方式。

final 类

有时你会想要一个类成为某个主题的最终结论。通过将整个类标记为 final，你就可以实现这一目标：

```
class Widget {...}
final class UltimateWidget : Widget {...}
class PostUltimateWidget : UltimateWidget {...} // 出错！不能从 final 类继承
```

自在读书 www.zizidiary.com

final 类不能再被继承——在继承层次结构中，它就是“叶子”。有时，这是一种很重要的设计策略。很显然，final 类的所有方法默认都为 final，因为不可能会出现改写的情况。

final 类的一个很有意思的副作用就是保证了实现的坚实性。使用 final 类的客户端代码就可以确信所给类的方法都是已知的实现，并且这些方法的效果保证不会被某个子类调整。

6.7　封装

面向对象设计以及其他设计技术的典型特征便是封装（encapsulation）。对象将它们的实现细节封装起来，并只对外公开定义明确的接口。这样，对象便可以自由地更改各种实现细节，同时不会为客户代码造成破坏。这种做法使解耦程度更高，而依赖程度更小，同时也验证了这样一句格言：每一种设计方法，至始至终都旨在对依赖关系的管理。

还有，封装就是信息隐藏（information hiding）的表现，这是软件设计过程中的一种通用哲学。信息隐藏规定：在应用程序里的各种模块化元素应该把重点放在定义和使用彼此通信的抽象接口上面，同时将如何实现这些接口的细节隐藏起来。通常情况下，这些细节都与数据布局相关。正因为如此，“数据隐藏”也成了一个常见的概念。但是，单纯关注数据隐藏则会错失部分关键点，因为一个组件可能隐藏各式各样的信息，如设计决策或算法策略。

在今天，封装听起来很有吸引力，或许甚至有些理所当然，但这很大程度上都是经验积累的结果。在早些时候，事情可没那么明确。要知道，既然信息明摆着是件好事，拥有越多越好，那为什么还要隐藏它呢？

早在 20 世纪 60 年代，Fred Brooks（巨作《人月神话》的作者）就提倡使用透明性、白盒、“所有人都了解所有事”的方法来进行软件设计。在他的管理下，构建 OS/360 操作系统的团队通过一种复杂的硬拷贝注释的方法接收到了关于常规基础内容的项目的所有设计细节的文档[13，第 7 章]。该项目是成功的，但认为透明度起到了积极的作用，甚至自圆其说地认为通过集中式的管理，将项目的风险降到了最低，则是站不脚的。David L. Parnas[44]的一篇具有革命性的论文在社区知识彻底建立起了信息隐藏的概念。Brooks 本人在 1995 年解释说，他所倡导的透明度是《人月神话》唯一未能经受住时间考验的主元素。不过早在 1972 年，信息隐藏的概念仍然存在很大争议，这一点可以通过一位评论家对于 Parnas 的革命性论文的评论得到验证：“显然，Parnas 并不知道他在说什么，因为没人这样做。”可笑的是，只隔了 10 年时间潮流便开始迅速逆转，同一篇论文则几乎不被重视：“Parnas 只是记录下了所有优秀的程序员随时都会做的事情。”[32，第 138 页]

回到 D 语言所支持的封装，你可以在任何类型、数据、函数或方法的声明前缀里加上下面提供的 5 个访问说明符中的某一个。我们先从最封闭的说明符开始，然后一路向着声名狼藉的说明符前进。

自在读书 www.zizidiary.com

6.7.1 private

private（私有的）可用于类级、所有类的外层（模块级）以及结构的内部（参见第 7 章）。在所有这些环境里，private 都具有相同的能力：将符号访问限制在当前模块（文件）中。

这种行为与其他语言有所不同，这些语言只会将对私有符号的访问限制于当前类。但是，让模块级也成为私有的，其效果则与 D 语言的通用保护方法是一致的——这里的保护单位与操作系统的保护单位（文件和目录）是相同的。文件级保护的好处在于它有助于将小的、紧密耦合的且负有给定职责的实体集中到一起。如果需要进行类级的保护，则可以简单地将这个类放置到一个它自己独占的文件里。

6.7.2 package

package（打包的）可用于类级、所有类的外层（模块级）以及结构的内部（参见第 7 章）。在所有这些环境里，package 引入了目录级的保护：与当前模块同一个目录里的所有文件都可以访问这个符号。当前模块所在目录的子目录和父目录则没有这样的特权。

6.7.3 protected

访问指示符 protected（受保护的）只在类里才有意义，在模块级是没意义的。当访问指示符 protected 用于某个类 C 的内部时，它所表示的是仅限于定义 C 的那个模块和 C 的后代才能访问这个声明的符号，不管它们所在的模块是哪一个。例如：

```
class C {
    // x 仅在本文件内才可访问
    private int x;
    // 此文件加上所有直接或间接继承自 C 类的类都可以调用 setX()
    protected void setX(int x) { this.x = x; }
    // 大家都可以调用
    public int getX() { return x; }
}
```

同样地，protected 访问允许具有传递性——它不仅能传递到孩子，也能传递到所有完全继承自使用 protected 的那个类的后代。就访问权限而言，protected 算是相当大方的。

6.7.4 public

public（公共的）访问表示的是所声明的符号可以在应用程序里自由地进行访问。应用程序需要做的就是获得这个符号的可见性，最常用的方式就是导入定义它的那个模块。

自在读书 www.zizidiary.com

在 D 语言里，public 是默认的访问级别。因为声明顺序是不起作用的，因此一种比较好的风格就是将模块或类的可见接口放置在最前面，然后使用 private:来限制访问，并继续进行定义。这样，客户端只需要查看文件或类的开始部分便可以了解到它的可访问实体。

6.7.5 export

似乎到这里，public 就应该是访问级别的底线了，即所有级别中最宽松的那个。然而，D 语言还定义了一个更宽松的访问，即 export（导出）。当将 export 与符号一起使用时，便可以从定义它的那个程序外部访问这个符号。共享库就属于这种情形，它会向外界公开其接口。编译器会执行系统相关的步骤：请求被导出的某个符号，通常包含了该符号的一个特殊命名约定。目前，D 语言还没定义出复杂的可用于动态加载的基础设施，因此在某种程度上 export 还仅仅是一张正等待更多扩展支持的存根。

6.7.6 封装到什么程度

有一个有趣的问题我们该自问一下：如何比较这 5 种访问指示符？例如，我们已承认信息隐藏是件好事，由此得出 private 更“优于”protected，是合情合理的，因为它的限制更为严格。继续沿着这条思路，我们还可能会认为 protected 比 public 更好（瞎扯，public 设置的障碍很低，export 更别提了）。但是，目前尚不清楚如何将 protected 同 package 进行比较。最为重要的是，这样一种定性分析并没有给出一个明确的说法。例如，当一个设计决定放宽对某个符号的限制时，它到底需要封装到什么程度。protected 更接近 private，还是更接近 public，还是刚好处于中间的位置呢？衡量标准究竟是什么呢？

早在 1999 年 12 月，当其他所有人都在担心千年虫（Y2K）时，Scott Meyers 则在焦虑封装的事，更确切地说，是在考虑能让其最大化的编码技术。在他随后的文章里[41]，为确定实体封装的“量”（amount），Meyers 提出了一个简单的准则：如果我们要改变该实体，会有多少代码受到影响呢？影响代码越少，那么封装就实现得越好。

使用一种方法来测量封装的程度能将事情说得更清楚一些。目前还没有哪个指标可以满足这个要求，有一种常见看法就是：private 很好，public 很糟，而 protected 则介于两者之间。因为乐观是人的天性，所以 protected 已经成为很多人常用的一种感觉不错的保护级别，这有点像“克制性饮酒”[①]。

在评估封装程度时，我们可以运用的另一层面是“控制”（control），即对那些可能受到更改影响的代码进行干预。你知道（或者你可以轻易地找到）受更改影响的代码有哪些吗？你有权利去修改这些代码吗？其他人可以往它里面添加代码吗？这些问题的答案就定义出了控制的程度。

① 原文为“drinking responsibly”，直译即为“尽责饮酒”的意思。其潜在的含义是：在享受饮酒乐趣的同时，不要让酒精控制到你。　——译者注

自在读书 www.zizidiary.com

首先，说说 private。修改一个 private 符号只会影响一个文件。一个源文件拥有上千个顺序行。因为大型文件（如有 10 000 行）会变得更难管理，所以常见的都是一些小文件。基于你正在对它进行更改这样一个事实，所以你能控制到这个文件；并且通过文件属性、版本控制或者团队编码标准，你也能够轻易地限制他人访问它。因此，private 提供了非常优秀的封装：受影响代码少，而且代码受到良好的控制。

当你使用 package 级的访问时，在同一个目录里的所有文件都将受到更改的影响。我们可以估算出分组成各种包的那些文件所拥有代码行的数量级（如可以认为一个包包含 10 个模块算是合理的）。相应地，将这些包符号混合在一起是需要成本的：更改会影响一定数量级的代码，这比针对 private 符号的类似更改所产生的影响要多。不过，幸运的是你可以很好地控制受影响的代码；同样的道理，因为操作系统和各种版本控制工具都支持在目录级控制添加和更改文件。

但是，protected 所保护的内容比表面上看起来要少。首先，protected 标志着要与 private 和 package 清楚地划清界线：处于程序里的任何地方的任何类都可以访问受保护的符号，方法便是继承那个定义它的类的后代。你不必对继承进行过细的控制，只需将全部属性设置为 final 或者全都不为 final 即可。由此可见，如果你夹杂了一个 protected 符号，那么你将影响不计其数的代码。更糟的是，你不仅没有能力限制别人继承你的类，你也会破坏连你自己都没权利去修复的代码（如更改一个库的符号就会影响到其他地方的应用程序）。现实是脆弱和残酷的：一旦你脱离了 private 和 package，你便来到了荒郊野外。使用 protected 很难提供任何保护。

当你更改一个 protected 符号时需要检查多少代码呢？它将是定义此符号的那个类的所有后代。合理的范围是其数量级超过一个包的大小，或者成千上万行。这时，源代码索引工具和类后代跟踪工具就能派上很大的用场；不过最后还有一点，更改 protected 符号会潜在地影响大量代码。

在控制方面，使用 public 不会有太多变化，但对于可能受影响的代码量，它又多增加了一个数量级。现在影响到的不仅仅是一个类的后代，而是整个应用程序的代码。最后，export 让情况出现了一种很有意思的转变——所有二进制应用程序都以二进制库的形式使用你的代码。这样，你不仅能看到你无法修改的代码，甚至你也不用去看代码，因为它可能不是以源代码形式提供的。

图 6-3 针对每一种访问指示符，绘制出了它们会潜在影响到的代码行数，显示的是一个近似范围。当然，这个量是估算的，并且可能与实际相差很远，但大致的比例应该不会受到什么影响。纵轴使用了对数刻度，而步长表示的是一种线性趋势，因此当你每次降低一个刻度的访问保护时，你就必须相应地付出约 10 倍的努力。往上指的箭头表明的是失去的对受影响代码的控制。这会得出一个必然的结论，即 protected 并不在 private 和 public 之间——它比较像 public，因此你应该以此对待它（也就是说，需要当心隔代遗传的问题）。

自在读书
www.zizidiary.com

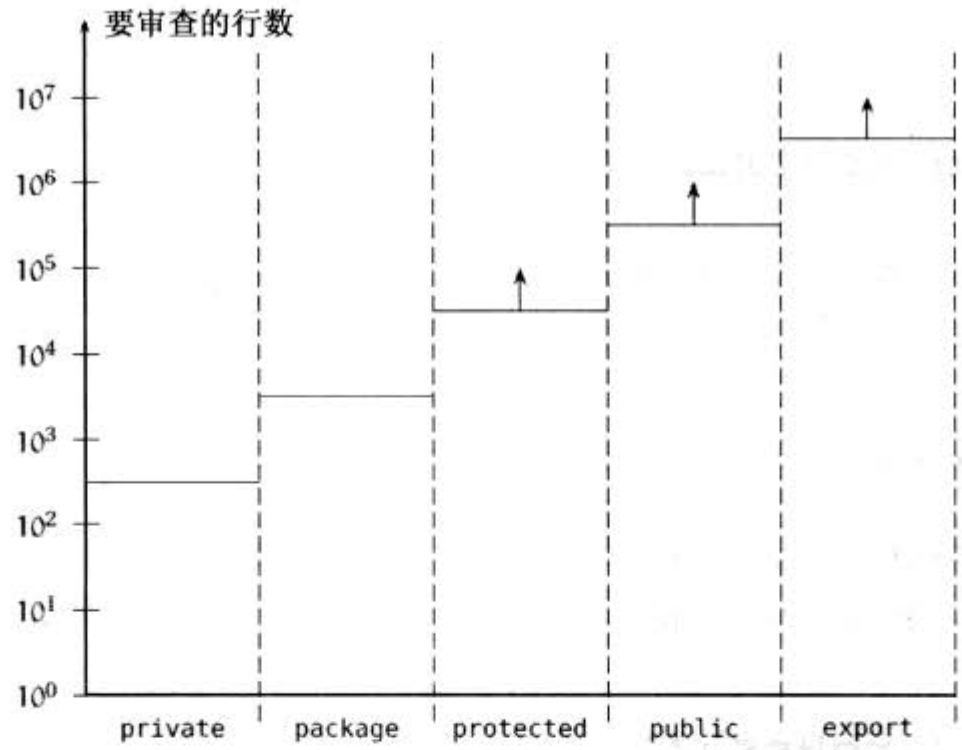


图 6-3　对于每一种保护级别，通过调整其符号可潜在影响代码行的范围估计。纵轴使用了对数刻度，因此封装的每一个放松都会让事情变坏一个数量级。往上指的箭头表征的事实是：受 protected、public 和 export 影响的代码的数量不受维护该符号的代码编写人员的控制

6.8　一统天下的根类

有些语言为所有其他类定义了一个根类，D 语言也位列其中。所有类的根类为 Object。如果像下面这样定义了一个类：

```
class C {
    ...
}
```

那么对于编译器来说，实际的内容就像下面这个样子：

```
class C : Object {
    ...
}
```

除了上面的自动重写外，Object 并没什么特殊之处——它就是一个类，别无他样。编译器实现会将它定义在一个叫做 object.di 或 object.d 的模块里，此模块会自动导入你需要编译的每个模块里。在 D 语言编译器实现有的目录里找找，你应该很容易找到此模块，然后可以查看一下。

让所有类都拥有一个根类，这样做有很多好处。其中一个明显的好处就是 Object 可以引入一些通用的方法。下面是略微有些简化的 Object 类的定义：

```
class Object {
    string toString();
    size_t toHash();
    bool opEquals(Object rhs);
    int opCmp(Object rhs);
    static Object factory(string classname);
}
```

自在读书 www.zizidiary.com

下面我们进一步了解一下其中每一个符号的语义。

6.8.1 string toString()

它返回的是对象的文本表示。默认情况下它会返回类名：

```
// 文件 test.d
class Widget {}
unittest {
   assert((new Widget).toString() == "test.Widget");
}
```

请注意，类的名称包含了定义该类的那个模块的名字。默认情况下，模块的名字与文件名是一样的，默认名字也可以使用 module 声明进行更改（参见 11.1.8 节）。

6.8.2 size_t toHash()

它返回的是对象的散列值，其形式是一个无符号整数值（在 32 位机器上为 32 位，而在 64 位机器上为 64 位）。默认情况下，散列值的计算是通过对象的按位表示进行的。散列值是对象的一个简洁但不精确的摘要。有一项重要的要求是一致性（consistency）：针对同一个引用对象在其状态未发生更改的时候，如果两次调用其 toHash，则应该返回相同的值。同时，散列值相等的两个对象也一定是相等的，而对于两个有差异的（不相等的）对象，它们的散列值是不可能相等的。下一节将详细讨论如何定义对象相等。

6.8.3 bool opEquals(Object rhs)

如果 this 认为 rhs 与它自己是相等的，它便会返回 true。这种奇怪的格式是故意设计的。根据 Java 的类似函数 equals 的经验：在存在继承的情况下，定义相等性会出现一些很奇妙的问题。因此，D 语言采取了一种相对巧妙的方式来解决这个问题。

首先，对象的相等概念已经存在：当你通过表达式 a1 is a2（参见 2.3.4.3 节）比较两个类对象 a1 和 a2 的引用时，当且仅当 a1 和 a2 都引用同一个对象，才能得到 true，如图 6-1 所示。这个对象相等的概念很容易理解，但受限过多而且没什么用处。通常，对于两个实际上不相同的对象，如果它们拥有相同的状态，那么也应被视为相等。在 D 语言里，逻辑相等可以通过“==”和“!=”操作符来进行计算。这里解释一下它们是如何工作的。

假设你针对表达式‹lhs›和‹rhs›编写了表达式‹lhs› == ‹rhs›。那么，如果‹lhs›和‹rhs›中至少有一个拥有自定义类型，那么编译器会将这个比较重写成 object.opEquals(‹lhs›, ‹rhs›)。类似地，‹lhs› != ‹rhs›会被重写成!object.opEquals(‹lhs›, ‹rhs›)。回顾一下本节的开始部分：object 是一个核心模块，它由 D 语言的编译器实现定义，并隐式地导入到你构建的任何模块里。因此，这些比较会被重写成对一个自由函数的调用，该

自在读书 www.zizidiary.com

函数由语言实现提供，并位于 object 模块内。

对象之间的相等关系是要求服从某些不变式，而 object.opEquals(‹lhs›, ‹rhs›) 在很大程度上确保了它的正确性。首先，空引用(null)一定会进行相等比较。其次，对于任何 3 个非空引用 x、y 和 z，下面的断言必须要始终为真：

```
// null 引用是单独的；不会存在非 null 对象等于 null 的情况
assert(x != null);
// 自反性
assert(x == x);
// 对称性
assert((x == y) == (y == x));
// 传递性
if (x == y && y == z) assert(x == z);
// 与 toHash 的关系
if (x == y) assert(x.toHash() == y.toHash());
```

有一个对 opEquals 更为微妙的要求是一致性（consistency）：针对相同的引用，在底层对象没有被更改的情况下，两次计算相等性必须要返回相同的结果。

典型的 object.opEquals 实现会先排除一些简单的或已退化的情况，然后转移到对象的成员实现的相等比较版本。object.opEquals 看起来有些像下面这个样子：

```
// 在系统模块 object.d 内部
bool opEquals(Object lhs, Object rhs) {
    // 如果是同一对象的别名或者两都都为 null，则相等
    if (lhs is rhs) return true;
    // 如果有一个为 null，则不相等
    if (lhs is null || rhs is null) return false;
    // 如果正好是相等的类型，则其中一个调用方法 opEquals
    if (typeid(lhs) == typeid(rhs)) return lhs.opEquals(rhs);
    // 如果是一般情况，则对称调用方法 opEquals
    return lhs.opEquals(rhs) && rhs.opEquals(lhs);
}
```

首先，如果两个引用都指向同一个对象或者都为 null，那么结果肯定为 true（确保自反性）。其次，一旦确立这些对象是不同的，如果它们当中有一个为 null，那么比较结果为 false（确保 null 的特殊性）。再次，测试检查这两个对象是否拥有相同的类型，如果它们相同，则转交给 lhs.opEquals(rhs) 处理。更有意思的部分是最后一行里的那个双重运算。难道只调用一个不行吗？

回想一下开始时对 opEquals 方法的描述（有点含糊）：“如果 this 认为 rhs 与它相等，它就会返回 true。”这个定义只关心 this，并未考虑 rhs 可能会有“异意”。为达成一致协议，还必须进行一次握手——两个对象都必须要明确地回应这样一个问题：你认为这个对象与你相等吗？正如 Joshua Bloch 在他的书籍 *Effective Java* 中[9]以及 Tal Cohen 随后在一篇文章[17]中所指出的：关于相等性的分歧可能只是一个学术问题，但在存在继承的地方它们相当普遍。我们要重申这一观点。

回到与图形用户界面相关的示例，假设要定义一个图形小部件，它可以放置在一个窗口中：

```
class Rectangle {...}
class Window {...}
class Widget {
    private Window parent;
    private Rectangle position;
```

自在读书 www.zizidiary.com

```
    ... // Widget 相关的函数
}
```

然后定义一个 TextWidget 类，它是一个可以显示某种文本的小部件。

```
class TextWidget : Widget {
    private string text;
    ...
}
```

对于这两个类，我们要如何实现 opEquals 呢？就 Widget 而言，另一个拥有相同状态的 Widget 是相等的：

```
// 在类 Widget 内部
override bool opEquals(Object rhs) {
    // 另一个必须为 Widget
    auto that = cast(Widget) rhs;
    if (!that) return false;
    // 比较所有的状态
    return parent == that.parent
        && position == that.position;
}
```

表达式 cast(Widget) rhs 会尝试将 rhs 恢复成 Widget。如果 rhs 为 null，或者 rhs 实际的动态类型不是 Widget 或 Widget 的子类，那么这个 cast 表达式会返回 null。

类 TextWidget 拥有更多存在差异的相等性概念，因为这个比较的右边必须是一个 TextWidget，并带有相同的文本。

```
// 在类 TextWidget 内部
override bool opEquals(Object rhs) {
    // 另一个必须为 TextWidget
    auto that = cast(TextWidget) rhs;
    if (!that) return false;
    // 比较所有相关状态
    return super.opEquals(that) && text == that.text;
}
```

现在假设有一个类型为 TextWidget 的 tw 与一个拥有相同的位置和父窗口的类型为 Widget 的 w，它们出现了叠加。就 w 而言，tw 与它是相等的。但从 tw 来看，却是不平等，因为 w 不是一个 TextWidget。如果我们承认 w == tw，但 tw != w，那么会破坏相等操作符的对称性。为恢复自反性，我们可以考虑让 TextWiget 不那么严格：在 TextWidget.opEquals 内部，如果进行比较的是 Widget，而非 TextWidget，那么这个比较就遵从 Widget 的相等概念。具体实现看起来有点像下面这个样子：

```
// TextWidget 的另一个 opEquals 版本 -- 坏的版本
override bool opEquals(Object rhs) {
    // 另一个必须至少为 Widget
    auto that = cast(Widget) rhs;
    if (!that) return false;
    // 它们会当做 Widgets 进行比较吗？如果不是，那我们就结束
    if (!super.opEquals(that)) return false;
    // 它是 TextWidget 吗？
    auto that2 = cast(TextWidget) rhs;
    // 如果不是，那我就成功结束比较
    if (!that2) return true;
    // 当做 TextWidgets 进行比较
    return text == that2.text;
}
```

自在读书 www.zizidiary.com

唉，建议 TextWidget 进行的调节尝试并不正确。问题在于，现在破坏了比较的传递性：很容易创建两个类型为 TextWidgets 的 tw2 和 tw1，它们是不相同的（包含不同的文本），但同时却与一个简单的 Widget 对象 w 相等。这造成的情形是 tw1 == w，并且 tw2 == w，但 tw1 != tw2。

因此，在一般情况下，比较必须要在两个方向进行——比较的每一端都必须要认可相等性。有一个好消息，即自由函数 object.opEquals(Object, Object) 可以避免这种握手，只要两边涉及的对象拥有完全相同的类型，而对于其他的情况彼此能进行调用即可。

6.8.4 int opCmp(Object rhs)

它实现了一个三相排序比较（three-way ordering comparison）。在关联数组里，对于将对象当做键来使用的情形，这种比较是很有必要的。如果 this 小于 rhs，它会返回一个不定负数；如果 rhs 小于 this，则返回一个不定正数；如果 this 与 rhs 被认为是无序，则返回 0。与 opEquals 相似，opCmp 很少被显式调用。多数时候，是通过 a < b、a <= b、a > b 以及 a >= b 这几个表达式当中的某一个来隐式调用它。

通过全局的 object.opCmp 定义（此定义负责涉及的两个对象之间的通信），这次重写遵循的协议与 opEquals 很相似。对于<、<=、>、>=这些操作符中的每一个，D 语言编译器都会将表达式 a ‹op› b 重写成 object.opCmp(a, b) ‹op› 0。例如，a < b 会变成 object.opCmp(a, b) < 0。

opCmp 的实现是可选的。默认的 Object.opCmp 实现会抛出一个异常。如果要实现它，那么 opCmp 必须要严格弱序（strict weak order），即是说，对于任何非空引用 x、y 和 z，它必须满足以下不变式。

```
// 1. 自反性
assert(x.opCmp(x) == 0);
// 2. 符号的传递性
if (x.opCmp(y) < 0 && y.opCmp(z) < 0) assert(x.opCmp(z) < 0);
// 3. 与 0 相等的传递性
if ((x.opCmp(y) == 0 && y.opCmp(z) == 0) assert(x.opCmp(z) == 0);
```

上述规则可能显得有点奇怪，因为它们使用了大家不太熟悉的“三相比较”的概念来表达这些公理。如果我们使用“<”来重写它们，就能得到在数学上熟悉的严格弱序特性：

```
// 1. '<'的非自反性
assert(!(x < x));
// 2. '<'传递性
if (x < y && y < z) assert(x < z);
// 3. '!(x < y) && !(y < x)'的传递性
if (!(x < y) && !(y < x) && !(y < z) && !(z < y))
   assert(!(x < z) && !(z < x));
```

为了保证“<”严格弱序，第三个条件必不可少。如果没有它，“<”将成为一个偏序（partial order）。如果你侥幸获得一个偏序，那么只能进行一些有一定限制的使用；最有意义的算法需要严格弱序。如果想定义一个偏序，那么最好什么语法糖都别用，并定义出自己的与 opCmp 截

自在读书 www.zizidiary.com

然不同的命名函数。

请注意，上面的条件只关注了“<”，对其他的排序比较却未提及，这是因为后者仅仅是语法糖而已（x > y 等同于 y < x，而 x <= y 等同于!(y < x)，等等）。

有一种从非自反性和传递性派生出的特性，有时会与公理相混淆，它就是反对称性（antisymmetry）：x < y 即意味着!(y < x)，这一点很容易通过归谬法（reductio ad absurdum）进行验证：绝不存在 x 和 y，使得 x < y 和 y < x 同时为真；如果出现了这种情况，那么就可以根据“<”的传递性将 z 替换成 x，例如：

```
if (x < y && y < x) assert(x < x);
```

根据假设，这个被测试的条件为真，因此这里的断言会被检查。但因为非自反性的关系，这个断言绝不会通过，这样与假设便产生了矛盾。

除了上面的限制，opCmp 还必须与 opEquals 一致：

```
// 与 opEquals 的关系
if (x == y) assert(x <= y && y <= x);
```

opEquals 的关系很松弛：可能存在这样一些类，它们能让 x <= y 和 y <= x 同时为真，因此常识表明它们是相等的。但是，x == y 并不是确定的。举一个简单的例子，假设有一个类，它进行相等比较时遵照的是区分大小写的字符串匹配方法，而排序时却遵照不区分大小写的字符串匹配方法。

6.8.5 static Object factory(string className)

这是一个很有意思的方法，它允许你根据指定的类名创建一个对象。所涉及的类必须接受不带参数的构造操作；否则，factory 会抛出一个异常。现在来测试一下 factory。

```
// 文件 test.d
import std.stdio;

class MyClass {
   string s = "Hello, world!";
}
void main() {
   // 创建一个 Object
   auto obj1 = Object.factory("object.Object");
   assert(obj1);
   // 现在创建一个 MyClass
   auto obj2 = cast(MyClass) Object.factory("test.MyClass");
   writeln(obj2.s); // 输出"Hello, world!"
   // 对不存在的类执行 factory 时，会返回 null
   auto obj3 = Object.factory("Nonexistent");
   assert(!obj3);
}
```

具有通过一个字符串创建对象的能力对于很多常见用法（如工厂设计模式[27，第 23 章]和对象序列化）非常有用。表面看来，下面的代码没有任何错误：

```
void widgetize() {
   Widget w = new Widget;
   ... 使用 w ...
}
```

自在读书 www.zizidiary.com

但是，后来你改变了主意，并决定为手头上的任务派生一个 TextWidget 类，因此你需要将上面的代码更改成这个样子：

```
void widgetize() {
   Widget w = new TextWidget;
   ... 使用 w ...
}
```

问题在于你需要“更改代码”。为了新功能而对代码大动干戈是很不好的，因为它很容易破坏现有的功能。理想情况下，你只需要将代码添加到新增功能里，因而才能拥有这样的自信：现有的代码能够像往常一样继续工作。这是可重写函数最能发挥作用的时候——通过调整特定的和良好定义的定制点，你便可以对代码进行定制，而事实上你又未曾改变它。Meyer 给这个概念取了一个听起来很富有禅意的名字，即“开/闭原则”[40]：一个类（广义上指的是一个封装单位）对扩展应当是开放的，而对修改是封闭的。这里的操作符 new 对所有情形都能准确地工作，因为如果想调整它的行为，更改 w 的初始值即可。一种更好的解决方案是从外面通过传递类名字来创建这个类，从而将 widgetize 从只能需要使用某个小部件的选择中解耦出来：

```
void widgetize(string widgetClass) {
   Widget w = cast(Widget) Object.factory(widgetClass);
   ... 使用 w ...
}
```

现在，widgetize 不用再负责选择该用哪个具体的小部件。也有其他一些方式可以实现灵活的对象构造，它们从不同方向探索着设计空间。如果想要深入讨论这个话题，则请仔细阅读那篇备受关注的名为“Java's new considered harmful”[4]的文章。

6.9　接口

通常，类对象会包含状态，并且会定义处理这些状态的方法。因此，一个给定的对象既是一个对外界的接口（通过 public 方法），同时也是一个封装好的对该接口的实现。

有时，将接口的概念从实现中分离出来是很有用的。当想要在大程序的各个不同部分之间定义通信时，这种做法会更加有用。一个想要操作 Widget 的函数完全只是关心 Widget 的接口——Widget 的实现对封装的真正定义来说是无关紧要的。由此引出了完全抽象接口的概念，即它只由类必须要实现的方法构成，但不包含任何实现。这种实体被称作“接口”（interface）。

D 语言的接口定义看起来有点像被限制后的类定义。除将关键字 class 更换成 interface 外，接口定义还需要服从某些限制。在接口里，不能定义任何非静态数据，并且你不能为任何可改写的函数提供实现。在接口内部，定义静态数据和带有实现的最终函数是合法。例如：

```
interface Transmogrifier {
   void transmogrify();
   void untransmogrify();
   final void thereAndBack() {
      transmogrify();
      untransmogrify();
   }
}
```

自在读书
www.zizidiary.com

下面是使用 Transmogrifier 的函数需要编译的所有内容。例如：

```
void aDayInLife(Transmogrifier device, string mood) {
   if (mood == "play") {
      device.transmogrify();
      play();
      device.untransmogrify();
   } else if (mood == "experiment") {
      device.thereAndBack();
   }
}
```

当然，鉴于目前还未定义 Transmogrifier 的原语，因此也就没办法调用 aDayInLife。现在来创建此接口的实现：

```
class CardboardBox : Transmogrifier {
   override void transmogrify() {
      // 进入盒子
      ...
   }
   override void untransmogrify() {
      // 从盒子出去
      ...
   }
}
```

这个接口的实现使用的语法与常规继承一样。由于有了 CardboardBox，我们现在就可以发起如下的调用：

```
aDayInLife(new CardboardBox, "play");
```

一个接口的任何实现都是该接口的子类型，因此实现会自动转换成接口。只需简单地传递一个 CardboardBox 对象来代替 aDayInLife 所期望的 Transmogrifier，我们便使用了接口。

6.9.1　非虚接口（NVI）惯用法

Transmogrifier 接口有 final 函数，这个可能看起来有一点陌生。关于抽象的赞美之词还有用吗？是功能仍未实现吗？如果接口就是抽象，那么它理所当然不应定义实现。

在 2001 年，Herb Sutter 写了一篇文章[52]，在里面他提到了一个很有意思的发现，而后来他又在一本书[55，第 39 条]中对其进行了重新整理。接口定义的可改写方法（如示例里的 transmogrify 和 untransmogrify）扮演的是一种双重角色。首先，它们既是接口自身的元素，也是调用代码为完成任务而使用的方法。其次，这种方法也是一个定制点，因为那也是继承类直接定义的内容。Sutter 认为，对这样两类方法进行区分可能有用：接口可以定义某些底层的抽象方法，并在以后实现它们；另外也可以定义高级别的、可见的、不可改写的方法，客户端代码可以使用它们。这两个集合可以出现重叠，但把它们当成是相等的则适得其反。

将客户端看到的内容与实现者所定义的内容进行区分能带来很多好处。这种方法允许你设计出兼有实现友好和客户端友好的接口。接口将实现的需求与客户端的需求进行合并，必须要在两者的需求之间进行妥协。过多地关注实现则会导致接口过于呆板、累赘和底层，这种接口

自在读书 www.zizidiary.com

会造成客户端代码的重复；而过多地关注客户端代码又会导致接口过大、松散和冗余，这种接口除了提供原语外，还要能提供一些便捷函数。使用非虚接口（Non-Virtual Interface，NVI）惯用法，可以让双方都感到满意。例如，Transmogrifier.thereAndBack 为调用者提供了一个便捷函数，而此函数是以原语的形式提供的。

这个新出现的特性与模板方法模式（它使用定制好的中间步骤指定一些比较固定的高级操作）是一致的；它看起来特殊是因为它有自己的名字，即非虚接口（NVI）。遗憾的是，尽管 NVI 已经成为一个广泛使用的模式，但它主要是优秀设计师中的一种惯例，很少作为一种语言辅助手段来强制保持设计的一致性。支持 NVI 的语言一直很少，主要原因在于一些流行的 OOP 语言的定义要早于(且造就了)对导致 NVI 的 OOP 设计的更好的理解。因此 Java 并不支持 NVI，C#也基本上不支持它（只是将它当做一种设计准则广泛使用），而 C++具有比较好的辅助协定支持，但是它对调用者和实现者都未提供强有力的保证。

当接口使用访问说明符时，D 语言通过提供特定的保证完全支持 NVI。假如 Transmogrifier 的作者担心出现不正确的用法——如果有人调用了 transmogrify，然后忘记了调用 untransmogrify 会怎么样呢？我们只将 thereAndBack 公开给客户端，同时仍然要求实现定义 transmogrify 和 untransmogrify：

```
interface Transmogrifier {
    // 客户接口
    final void thereAndBack() {
        transmogrify();
        untransmogrify();
    }
    // 实现接口
private:
    void transmogrify();
    void untransmogrify();
}
```

这里的 Transmogrifier 接口让那两个原语变成了私有的。这种调整形成了一种很有意思的设计：实现 Transmogrifier 的类必须定义 transmogrify 和 untransmogrify，但却不可调用它们。事实上，在 Transmogrifier 模块之外没人能调用到这两个原语。间接调用它们的唯一方法是调用更高一层的 thereAndBack 例程，而这正是设计的意义所在：定义良好的访问点，以及围绕着这些访问点的结构良好的调用流程。D 语言将平时可能对这种保证产生破坏的行为拒之门外。例如，实现类不可能放宽 transmogrify 和 untransmogrify 的保护级别：

```
class CardboardBox : Transmogrifier {
    override private void transmogrify() { ... } // 正确
    override void untransmogrify() { ... }       // 错误!
        // 不能将 untransmogrify 的保护
        // 从 private 更改至 public!
}
```

当然，因为它毕竟是你的实现，所以如果需要，你也可以让原语变成公共的，但你必须换一个名字：

```
class CardboardBox : Transmogrifier {
    override private void transmogrify() { ... } // 正确
    override private void untransmogrify() {     // 正确
```

自在读书 www.zizidiary.com

```
        doUntransmogrify();
    }
    void doUntransmogrify() { ... }                    // 正确
}
```

现在 CardboardBox 的使用者可以调用 doUntransmogrify，它所做的事情与 untransmogrify 一样。但最重要的一点在于：具有特定名称和签名的 void untransmogrify() 不会直接暴露给实现类。所以客户端代码绝不会访问到以 private 名义提供的私有功能。如果某个实现希望定义和记录另一个函数，那是它自己的事。

D 语言保持 NVI 一致性的第二种方法是禁止劫持(hijacking)最终方法：Transmogrifier 的实现者不能再定义实际上是劫持 thereAndBack 的方法。例如：

```
class Broken : Transmogrifier {
    void thereAndBack() {
        // 为何不执行两次呢?
        this.Transmogrifier.thereAndBack();
        this.Transmogrifier.thereAndBack();
    }
    // 错误! 不能劫持 final 型方法 Transmogrifier.thereAndBack
    ...
}
```

如果允许这种隐藏方式，客户端即使知道 Broken 实现了 Transmogrifier，但在面对类型为 Broken 的对象 obj 时，它仍无法确信调用 obj.thereAndBack()。确切地说是不太确定 thereAndBack 的内容是否与 Transmogrifier 规定和记录的内容一致。当然，客户端代码可以调用 obj.Transmogrifier.thereAndBack()，以此保证调用路由的正确性，但这种靠注意力驱动的设计从来都不太受欢迎。要知道，一个好的设计不会在你警觉性降低的时候使用一些莫名其妙的行为来打击你。总结一下：如果接口定义了一个 public 函数，那么这个函数在其所有实现里都是可见的；如果这个实现同时还为 final，那么实现类便无法拦截对这个函数的调用。但是，只要不存在潜在的冲突，实现也可以定义具有相同名字的函数。例如：

```
class Good : Transmogrifier {
    void thereAndBack(uint times) {
        // 为何不多次执行它呢?
        foreach (i; 0 .. times) {
            thereAndBack();
        }
    }
    ...
}
```

上述情况是允许的，因为不存在混淆的风险：一个调用，要么可以拼写成 obj.thereAndBack()，从而转到 Transmogrify.thereAndBack；要么拼写成 obj.thereAndBack(n)，从而转到 Good.thereAndBack。也就是说，Good.thereAndBack 的实现不需要修饰其内部对同名接口函数的调用。

6.9.2　protected 原语

在接口里，让某个可改写的函数成为 private，更多时候是用于限制，而非满足需要。例

自在读书 www.zizidiary.com

如，它可以禁止接口实现调用父级函数，如下所示：

```
import std.Exception

class CardboardBox : Transmogrifier {
private:
   override void transmogrify() { ... }
   override void untransmogrify() { ... }
}
class FlippableCardboardBox : CardboardBox {
private:
   bool flipped;
   override void transmogrify() {
      enforce(!flipped, "Can't transmogrify: "
         "box is in time machine mode");
      super.transmogrify(); // 错误！不能调用私有方法 CardboardBox.transmogrify!
   }
}
```

当操作 CardboardBox 时，它不能表现得像 Transmogrifier 一样——众所周知，在这种情况下它就是一台时光穿梭机器。FlippableCardboardBox 实现了这一功能，但在正常路径里，它无法调用其父级的 transmogrify 方法。该怎么办呢？

有一种解决办法是使用上面演示的用于 doUntransmogrify 的重命名技巧，但如果需要对好几个方法都这样做，则重复内容较多。有一种更为简单的解决方法，即在 Transmogrifier 里放宽对两个可改写方法的访问限制，从 private 放宽至 protected。

```
interface Transmogrifier {
   final void thereAndBack() { ... }
protected:
   void transmogrify();
   void untransmogrify();
}
```

使用 protected 访问，接口实现现在便可以调用其父级实现。请注意，将保护级别加强也是非法的。如果某个接口定义了一个方法，则其实现不能为该方法声明更强的保护。例如，假设有 Transmogrifier，它将 transmogrify 和 untransmogrify 定义 protected 保护级，那么下面的代码便是错误的：

```
class BrokenInTwoWays : Transmogrifier {
  public void transmogrify(){ ... }          // 错误!
  private void untransmogrify(){ ... }       // 错误!
}
```

在一个实现里，同时满足对接口进行放宽和加强的要求，在技术上是可行的，但它很难满足优秀设计的目标要求。接口表达的是一种意图，阅读者只应该通过理解接口的定义来充分利用它，而不用去管实现者是否提供了静态类型。

6.9.3　选择性实现

有时，两个接口可能定义出存在歧义的 publicfinal 方法：

```
interface Timer {
   final void run() { ... }
   ...
}
```

自在读书 www.zizidiary.com

```
interface Application {
   final void run() { ... }
   ...
}
class TimedApp : Timer, Application {
   ... // 无法定义 run()
}
```

像这样的情况，TimedApp 便不能定义它自己的 run()，因为它实际上会劫持两个方法，而且两个劫持的情况肯定会比只有一个劫持的情况更糟。在 Timer 和 Application 之间消除任何一个的 final 方法都无助于解决这个问题，因为始终都有一个劫持在起作用。如果两个都为非 final，那么我们便有了一个很好的外形——TimedApp.run 同时实现 Timer.run 和 Application.run。

对于类型为 TimedApp 的 app，想要访问这些方法，则必须分别写成 app.Timer.run() 和 app.Application.run()，以对应 Timer 和 Application 里的版本。TimedApp 可以定义自己的函数来转向这些方法，只要它们不会劫持 run()。

6.10　抽象类

通常，基类不会为某些或全部方法提供任何切合实际的实现。有一种想法是将这种类转换成接口，但有时让它定义一些状态和非 final 方法（接口不支持这种特权），则能发挥更大的作用。抽象类（abstract class）这时就能派上用场：它与常规类很像，但通过将关键字 abstract 用于某个函数声明的前缀，它便可以不用实现这个函数。

作为演示，来看一个历史悠久的示例。它表征的是物体的形状层次结构，这些形状的物体在面向矢量的绘图程序里经常用到。此层次结构的根为类 Shape。所有形状都有一个边框矩形，因此 Shape 可能会想要将其定义成一个成员变量（这是接口无法办到的）。另一方面，Shape 还必须没有定义如 draw 那样的方法，因为它无法明确地实现它们。这些方法的实现应由 Shape 的后代负责。

```
class Rectangle {
   uint left, right, top, bottom;
}
```

```
class Shape {
   protected Rectangle _bounds;
   abstract void draw();
   bool overlaps(Shape that) {
      return _bounds.left <= that._bounds.right &&
         _bounds.right >= that._bounds.left &&
         _bounds.top <= that._bounds.bottom &&
         _bounds.bottom >= that._bounds.top;
   }
}
```

方法 draw 是抽象的（abstract），它包含了 3 层含义。首先，编译器不会要求 Shape 实现 draw。其次，编译器不允许实例化 Shape。最后，编译器不允许实例化任何（在祖先里直接或间接）未实现 draw 方法的 Shape 后代。这里的“直接或间接”指的是这种实现要求没有传递性。例如，定义一个 RectangularShape 类，它继承自 Shape，并实现了 draw 方法，

自在读书 www.zizidiary.com

因此你便不用要求 RectangularShape 的后代再实现它。

如果编译器不期望实现抽象方法，并不表示在你想要抽象方法的时候也不能提供抽象方法。例如，你可以提供对 Shape.draw 的实现。客户端可以通过对调用进行显式修饰的方式调用它，如 this.Shape.draw()。

方法 overlaps 在被实现的同时也是可改写的，这个细节很有意思。默认情况下，overlaps 将两个形状的交集近似地当做是它们的边界矩形的交集来处理。对于大多数非矩形形状，这是不准确的，例如两个圆圈就可能不是重叠的，哪怕它们的边框存在重叠的情况也一样。

一个至少拥有一个抽象方法的类自身称为抽象类（abstract class）。如果类 RectangularShape 继承自抽象类 Shape，而同时没有改写 Shape 的所有抽象方法，那么 RectangularShape 也是抽象的，并且会将实现这些抽象方法的要求向下传递到 RectangularShape 的后代。此外，也允许 RectangularShape 引入新的抽象方法。例如：

```
class Shape {
   // 同上
   abstract void draw();

}
class RectangularShape : Shape {
   // 从 Shape 继承一个抽象方法同时引入更多方法
   abstract void drawFrame();
}
class Rectangle : RectangularShape {
   override void draw() { ... }
   // Rectangle 仍然是一个抽象类
}
class SolidRectangle : Rectangle {
   override void drawFrame() { ... }
   // SolidRectangle 是一个实体类：不再有需要实现的抽象函数
}
```

最有意思的是，一个类可以决定是否需要再次将一个函数设置为抽象的，即使它之前被改写过和已实现过也一样！下面的代码引入了一个抽象类，并由该类派生出了一个具体类，然后又由该具体类派生出一个抽象类，全部都是针对某一个方法。

```
class Abstract {
   abstract void fun();
}
class Concrete : Abstract {
   override void fun() { ... }
}
class BornAgainAbstract : Concrete {
   abstract override void fun();
}
```

你也可以终结一个抽象方法的实现……但很显然，你不能定义既为 abstract 也为 final 的方法。

```
class UltimateShape : Shape {
   // 关于方法 draw 的最终改写
   override final void draw() { ... }
}
```

如果你想引入一组抽象方法，那么可以使用类似于保护说明符的方式（参见 6.7.1 节），重

自在读书 www.zizidiary.com

用关键字 abstract：

```
class QuiteAbstract {
   abstract {
      // 此作用域里的所有内容都是抽象的
      void fun();
      int gun();
      double hun(string);
   }
}
```

在 abstract 的作用域内没有办法关闭 abstract，因此下面的定义并不正确：

```
class NiceTry {
   abstract {
      void fun();
      final int gun(); // 错误!
      // 不能定义 final abstract 函数!
   }
}
```

你可以将 abstract 用做标签：

```
class Abstractissimo {
abstract:
   // 下面所有内容都是抽象的
   void fun();
   int gun();
   double hun(string);
}
```

一旦引入 abstract:，你便不可能关闭它。

最后，你还可以将整个类标记为 abstract：

```
abstract class AbstractByName {
   void fun() {}
   int gun() {}
   double hun(string) {}
}
```

随着上面对 abstract 用法的逐渐加强，似乎让整个类变成抽象的能发挥更大的作用，它让该类里的每一个单个方法都变成了抽象的。并非如此！它实际上会因过于粗糙而无法派上任何用场。在 class 前面的 abstract 所做的就是防止客户端代码实例化它——只能实例化由它派生的非抽象类。继续上面的 AbstractByName 示例：

```
unittest {
   auto obj = new AbstractByName; // 错误！不能将抽象类 AbstractByName 实例化!
}
class MakeItConcrete : AbstractByName {
}
unittest {
   auto obj = new MakeItConcrete; // 正确
}
```

6.11 嵌套类

嵌套类是一个很有意思的特性，它值得特别注意。它们可用做许多重要特性的构件，如多重子类型化——在下一节会讨论到。

自在读书 www.zizidiary.com

在类的内部，可以定义另一个类：

```
class Outer {
    int x;
    void fun(int a) { ... }
    // 定义一个内层类
    class Inner {
        int y;
        void gun() {
            fun(x + y);
        }
    }
}
```

嵌套类就是一个普通的……等一下！Inner.gun 是如何访问到 Outer 的非静态成员变量和方法的？如果 Outer.Inner 只是一个传统的定义在 Outer 作用域内的类，那么它不可能获得数据和调用 Outer 对象的方法。事实上，这个对象来自何处呢？我们现在创建一个类型为 Outer.Inner 的对象，看看会发生什么事情：

```
unittest {
    // 无法工作
    auto obj = new Outer.Inner;
    obj.gun(); // 这个应该会让程序崩溃，因为
        // 看不到 Outer.x 或 Outer.fun——根本就没有外层 Outer!
}
```

因为这段代码只创建了一个 Outer.Inner，而没有创建 Outer，因此唯一被分配的数据是 Outer.Inner 所定义的内容（即 y），而非 Outer 所定义的（即 x）。

不过，令人惊讶的是，这个类定义可以编译，但单元测试 unittest 却无法编译。到底怎么回事？

首先，在没有 Outer 对象时，你永远不能创建 Inner 对象。这个限制很有意义，因为 Inner 可能会偷偷地访问 Outer 的状态和方法。下面展示了如何正确创建一个 Outer.Inner 对象：

```
unittest {
    Outer obj1 = new Outer;
    auto obj = obj1.new Inner; // 啊哈!
}
```

那个特殊的 new 语法表明有事情发生：在创建类型为 Outer.Inner 的对象之前，需要先创建类型为 Outer 的对象。该对象的引用（在我们的示例里为 obj1）是会被悄悄地存储在 Inner 对象内部的，存储形式为名叫 outer 的语言定义特性。然后，当要使用 Outer 的成员（如 x）时，这种访问会被重写成 this.outer.x。对于存储在内层对象里的那个隐藏引用，其初始化会正好发生在内层对象的构造函数被调用之前。因此，构造函数自身可以直接访问外部对象的成员。通过对 Outer/Inner 示例做些调整，我们可以实际地测试一下所有这些内容：

```
class Outer {
    int x;
    class Inner {
        int y;
        this() {
            x = 42;
            // x 或 this.outer.x 都是一回事
            assert(this.outer.x == 42);
        }
```

自在读书
www.zizidiary.com

```
    }
}
unittest {
    auto outer = new Outer;
    auto inner = outer.new Inner;
    assert(outer.x == 42); // inner 更改了 outer
}
```

如果在 Outer 的某个非静态成员函数里创建 Outer.Inner 对象，则没必要在 new 表达式的前缀里加上 this.，因为它是隐式的。例如：

```
class Outer {
    class Inner { ... }
    Inner _member;
    this() {
        _member = new Inner;                    // 等同于 this.new Inner
        assert(_member.outer is this);          // 检查链接
    }
}
```

6.11.1　函数嵌套类

将一个类嵌套于一个函数内部与将一个类嵌套于另一个类中有着惊人的相似之处。处于函数内部的类可以访问这个函数的参数和局部变量：

```
void fun(int x) {
    string y = "Hello";
    class Nested {
        double z;
        this() {
            // 访问参数
            x = 42;
            // 访问本地变量
            y = "world";
            // 访问自己的成员
            z = 0.5;
        }
    }
    auto n = new Nested;
    assert(x == 42);
    assert(y == "world");
    assert(n.z == 0.5);
    ...
}
```

当你拥有一个会返回类类型的函数，而你想继承该类型并想调整其行为时，嵌套在函数里的类就特别有用了。假设：

```
class Calculation {
    double result() {
        double n;
        ...
        return n;
    }
}

Calculation truncate(double limit) {
    assert(limit >= 0);
    class TruncatedCalculation : Calculation {
```

自在读书 www.zizidiary.com

```
        override double result() {
            auto r = super.result();
            if (r < -limit) r = -limit;
            else if (r > limit) r = limit;
            return r;
        }
    }
    return new TruncatedCalculation;
}
```

这个 truncate 函数会改写 Calculation 类的 result 方法，以便在一定范围内将它进行截断。对于 truncate 的工作，其中有一点非常微妙：请注意 result 的改写是如何使用 limit 参数的。只要 TruncatedCalculation 用在 truncate 里，这一点就没什么奇怪的，但 truncate 返回到外界的 TruncatedCalculation。有一个简单的问题，即在 truncate 返回之后，limit 存在于哪里呢？通常情况下，函数的参数和局部变量都位于栈上，并且在函数返回时会消失。在这里，在 truncate 返回之后，limit 被使用了，所以 limit 一定还存在于某个地方。否则，整个代码就会由于不安全访问已被释放的栈内存而崩溃。

这个示例借助于编译器，便能够很好地工作。每当编译函数时，编译器便搜索和检测非局部的“逃逸”情况，即这样一种状况：在函数返回之后，还有参数或局部变量仍在使用中。如果检测到这种“逃逸”情况，编译器便会将函数局部分配的状态（参数以及局部变量）从栈分配切换至动态分配。这样，truncate 的参数 limit 便可以在 truncate 返回之后成功存活，并被 TruncatedCalculation 使用。

6.11.2　静态嵌套类

我们需要面对这样的情况：嵌套类并不完全是它们看起来的样子。在表面上它们只是能定义在类或函数内部的普通类，但它们显然并不普通：特殊的 new 语法和语义、神奇的.outer 特性、修改后的查找规则——嵌套类绝对不同于普通类。

假如要在另一个类或函数内部定义一个友好类会带来什么后果？大量使用关键字 static 能进行补救：只需在类定义的前缀加上 static 即可。例如：

```
class Outer {
    static int s;
    int x;
    static class Ordinary {
        void fun() {
            writeln(s);                    // 正确，允许访问静态值
            writeln(x);                    // 错误！不成访问非静态成员 x！
        }
    }
}
unittest {
    auto obj = new Outer.Ordinary;  // 正确
}
```

静态的内层类就是一个普通类，它没有访问外层对象仅仅是因为那里没有这种对象。然而，由于其作用域的关系，静态内层类可以访问该封闭类的静态成员。

自在读书 www.zizidiary.com

6.11.3　匿名类

一个类定义在省略名称及其父类规范中的“:”之后，便会引入一个匿名类（anonymous class）。这种类必须始终（非静态地）嵌套在函数当中，其唯一的用途便是立刻创建一个新的对象：

```
class Widget {
   abstract uint width();
   abstract uint height();
}

Widget makeWidget(uint w, uint h) {
   return new class Widget {
      override uint width() { return w; }
      override uint height() { return h; }
      ...
   };
}
```

这种特性与匿名函数很像。创建匿名类相当于创建一个新的命名类并对它进行实例化，只是这两个步骤被合并成了一个。这种特性看似没什么，也没节省什么东西，但事实证明，许多设计都广泛使用它来将观察者与主题连接在一起[7]。

6.12　多重继承

D 语言使用的继承模型是类为单继承而接口为多重继承。这一点与 Java 和 C#的立场相似，与其他如 C++和 Eiffe 那样的语言所走的路线不同。

一个接口可以继承任意数量的接口。因为它不会实现任何可改写的函数，所以继承接口仅仅是一个增强后的接口，它要求的是其基接口的所有原语总和，外加它自己的一些可能的操作。假设：

```
interface DuplicativeTransmogrifier : Transmogrifier {
   Object duplicate(Object whatever);
}
```

接口 DuplicativeTransmogrifier 继承于 Transmogrifier，因此对于 DuplicativeTransmogrifier 的任何实现，除了要实现新引入的 duplicate 外，还必须实现 Transmogrifier 的所有原语。这种继承关系如期望的一样——你可以将 DuplicativeTransmogrifier 传递到所有期望是 Transmogrifier 的地方，但反之则不行。

通常情况下，一个接口可以继承任意数量的接口，并会得到全部要求的原语。同样，一个类也可以实现任意数量的接口。例如：

```
interface Observer {
   void notify(Object data);
   ...
}
interface VisualElement {
   void draw();
   ...
}
```

自在读书 www.zizidiary.com

```
interface Actor {
   void nudge();
   ...
}
interface VisualActor : Actor, VisualElement {
   void animate();
   ...
}
class Sprite : VisualActor, Observer {
   void draw() { ... }
   void animate() { ... }
   void nudge() { ... }
   void notify(Object data) { ... }
   ...
}
```

图 6-4 显示的是上面内容被编码后的继承层次结构。接口被编码为椭圆形，而类被编码为矩形。

现在我们定义一个 Sprite2 类。它的作者已经忘记了 VisualActor 其实就是一个 Actor，所以 Sprite2 除了继承 Observer 和 VisualActor 外，还直接继承了 Actor。图 6-5 显示的是最终的层次结构。

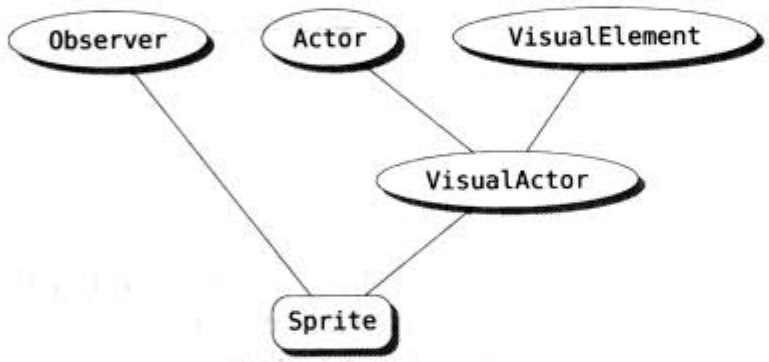


图 6-4　一个简单的继承层次结构，它表征的是接口的多重继承关系

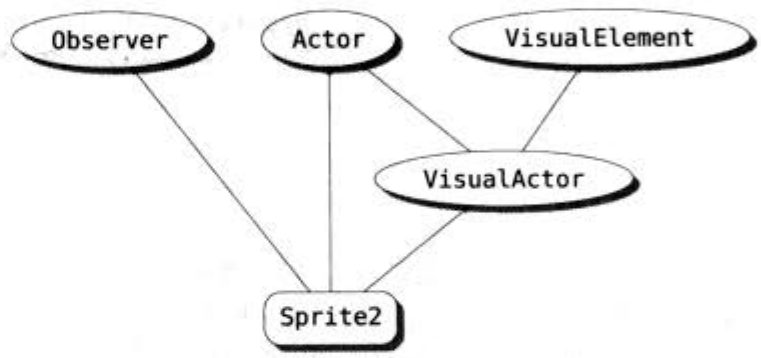


图 6-5　一个具有冗余路径（在本例中为 Sprite2 到 Actor 的链接）的继承层次结构。冗余的路径不是什么障碍，但消除它们往往比较烦琐，而消除的话可以让设计更清晰并减少对象的大小

在层次结构里立即便能看出来存在一条冗余路径，因为与某个接口存在一个直接连接，同时又间接继承了该接口。冗余路径不会造成什么特别的问题，但在多数实现里，它们会增加最终对象（在本例中为 Sprite2）的大小。

在某些情况下，通过两条路径继承了同一个接口，但却无法移除这两条路径。假设，我们先添加一个 ObservantActor 接口，它继承了 Observer 和 Actor：

```
interface ObservantActor : Observer, Actor {
   void setActive(bool active);
}
interface HyperObservantActor : ObservantActor {
   void setHyperActive(bool hyperActive);
}
```

然后，我们又定义 Sprite3 来实现 HyperObservantActor 和 VisualActor：

```
class Sprite3 : HyperObservantActor, VisualActor {
   override void notify(Object) { ... }
   override void setActive(bool) { ... }
   override void setHyperActive(bool) { ... }
   override void nudge() { ... }
```

自在读书 www.zizidiary.com

```
    override void animate() { ... }
    override void draw() { ... }
    ...
}
```

这个调整让事情发生了一点变化（图 6-6）。如果 Sprite3 想同时实现 HyperObservantActor 和 VisualActor，那么它最终会从不同的路径两次实现 Actor，这时就没办法消除它了。不过，幸运的是，编译器不会把这一点当成问题——它支持重复继承同一个接口。但是，重复继承同一个类则是不允许的，因为 D 语言不允许类出现任何形式的多重继承。

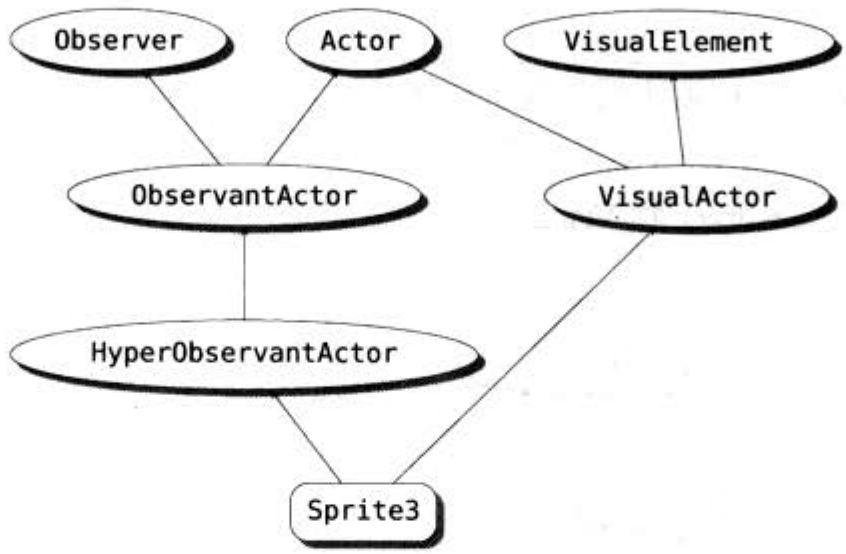


图 6-6　一个在两个节点之间（在本例中为 Sprite3 和 Actor）存在多条路径的层次结构。这种结构通常称为“钻石型继承层次结构”，因为当 HyperObservantActor 不存在时，Sprite3 和 Actor 之间的两条路径将描绘出一个钻石形状。在一般情况下，这种层次可能会呈现出各式各样的形状。其显著特征就是从一个节点到另一个节点存在多条间接路径

为什么会有这种差别呢？接口有什么特别之处，让它比类更适合于多重继承呢？完整的解释是很复杂的，不过扼要地讲，接口和类之间本质的差别在于：后者可能包含状态。更重要的是，类可能包含可修改的状态。相比之下，接口并没有自己的状态。对于每一个被实现的接口，都存在某种与之关联的“簿记”（在许多实现里，它是一个指向某张“虚拟表”的指针。其中的虚拟表，即为指向函数的指针数组），但在类里该指针对于某个接口的所有实例都是相同的，绝不会发生变化，并且是在编译器的控制之下。利用这些限制，编译器实际上会在这个类里植入多个这种簿记信息的副本，但该类绝对分辨不出来。

在历史上，已有了大量关于多重继承优缺点的争论。这种争论还没有统一，在短期内也不会统一。但有一件事是大家普遍认同的，即很难同时以简单、高效和有用的方式来实现多重继承。有些语言没有将效率作为主要考虑的内容，所以它们选择支持多重继承从而获得更多的表现力。另外一些语言则以获得最基本的性能（例如连续对象或快速函数调度）为前提，从而对设计的灵活性加以限制。Scala 的混合（mixin）是一种很有意思的设计，它既具备多重继承的大部分优势，又避开了它的弊端；其本质就是使用默认实现封装的接口。D 语言采用的方式是支持多重子类型化（subtyping），也就是说，没有继承的子类型化。现在来看看它是如何工作的。

自在读书
www.zizidiary.com

6.13　多重子类型化

我们继续构建 Shape 示例，并假设定义一些 Shape 对象，让它们可以存储在一个数据库里。我们找到了一个制作相当精巧的数据库持久性开发库，它能很好地满足我们的目标，只需要一点点的调整即可——它要求每一个可存储对象都继承一个名为 DBObject 的类。

```
class DBObject {
private:
   ... // 状态
public:
   void saveState() { ... }
   void loadState() { ... }
   ...
}
```

这种情况可以采用多种方式来构建模型，现在就来看看——如果语言的局限性不是一个问题，那么有一种很自然的方法，即定义一个 StorableShape 类，它“是一个” Shape，同时也“是一个” DBObject。所有形状的层次结构都是以 StorableShape 为根的。接着，当需要绘制一个 StorableShape 时，其外观应像一个 Shape。当想在数据库里来回操纵它时，它又会变得好似一个 DBObject。这就表明我们需要使用类的多重继承性，而这个在 D 语言里是被禁止的，因此我们需要寻找一种替代方案。

好在 D 语言可以借助一种通用且非常有用的机制——多重子类型化。一个类可以将自己指定为其他任何类的一个子类型，同时不用去继承这个类。所需要做的就只是指定一个 alias this 声明。一个最简单的示例有点像下面这个样子：

```
class StorableShape : Shape {
   private DBObject _store;
   alias _store this;
   this() {
      _store = new DBObject;
   }
   ...
}
```

StorableShape 继承了 Shape，于是它“是一个” Shape，但它也是一个 DBObject。每当需要从 StorableShape 转换到 DBObject 时，或者查找 StorableShape 的成员时，成员 _store 就会发挥作用。与 DBObject 相匹配的请求会被自动从 this 重定向至 this._store。例如：

```
unittest {
   auto s = new StorableShape;
   s.draw();            // 调用 Shape 的方法
   s.saveState();       // 调用 DBObject 的方法
                        // 可重写成 s._store.saveState()
   Shape sh = s;        // 正常的类型向上转换：从派生类型转换到基类型
   DBObject db = s;     // 可重写成 DBObject db = s._store
}
```

实际上，StorableShape 就是 DBObject 的一个子类型，而成员 _store 是 Storable

自在读书 www.zizidiary.com

Shape 对象的 DBObject 子对象。

一个类可以引入任意数量的 alias this 声明，这样它可以子类型化任意数量的类型。

多重子类型化场景里的方法改写

事情不可能如此简单，不是吗？它们的确不简单，因为 StorableShape 一直在进行欺骗。没错，利用 alias _store this 声明，StorableShape 在名义上成为了 DBShape，但 StorableShape 不能直接改写 DBShape 的任何方法。很显然，Shape 的方法可以像平常一样被改写，但在哪个地方才能改写 DBObject.saveState 呢？以一种伪子对象的形式返回 _store 是一种逃避的做法——事实上，关于与 StorableShape 外层对象相连接的_store，这里并没有什么内容，至少在我们实际改变它之前是没有的。现在来看看其中都包含了什么内容。

原来的 StorableShape 定义其确切的欺骗点是在使用 new DBObject 初始化_store 的时候。在这里，子对象_store 与 StorableShape 外层对象（它应该改写 DBObject 里的方法）之间的连接完全被断开。因此，我们需要做的便是在 StorableShape 内部定义一个新类 MyDBObject。这个类将存储一个对 StorableObject 外层对象的反向引用，并能改写任何需要被改写的方法。最后，在被改写方法的内部，MyDBObject 也能访问到整个 StorableObject，而这一切都像是成熟的多重继承在发挥作用。太了不起了！

如果在本章里“外层对象”（outer object）从一开始便被提请注意的话，那么你可能很早便能发现计算发展史上又一件最偶然的事件。嵌套类（参见 6.11 节）也能很好地满足多重子类型化的需要，你也许会认为它只是一个突发奇想的用来解决问题的小把戏。事实上，嵌套类（灵感源自 Java）要比 alias this 这一特性早好几年。

使用嵌套类让带有 alias this 的改写变得异常简单。在本例里，所有需要做的就只有定义一个嵌套类，让它继承 DBObject。在这个类里，你可以随意地改写任何方法，并且可以完全访问 DBObject 的 public 定义和 protected 定义，以及 StorableShape 的所有定义。如果还有比这更简单的事，那可能就是至少还有几个状态是非法的。

```
class StorableShape : Shape {
   private class MyDBObject : DBObject {
      override void saveState() {
         // 访问 DBObject 和 StorableShape
         ...
      }
   }
   private MyDBObject _store;
   alias _store this;
   this() {
      // 这里是建立链接的关键点
      _store = this.new MyDBObject;
   }
   ...
}
```

_store 可以访问到外层的 StorableShape 对象，这是相当关键的。正如 6.11 节所讨论

自在读书 www.zizidiary.com

的，创建嵌套类时会偷偷地在这个嵌套类里面存储这个外部对象（在我们的示例里指的是this）。符号 this.new MyDBObject 只是想要清楚地表明：this 决定着新 MyDBObject 对象的创建。事实上，在这里可以不显式使用 this.。

唯一的问题是，DBObject 的成员会将 StorableShape 的成员掩盖掉。例如，假设 DBObject 和 Shape 都定义了名叫_name 的成员变量：

```
class Shape {
   protected string _name;
   abstract void draw();
   ...
}
class DBObject {
   protected string _name;
   void saveState() { ... }
   void loadState() { ... }
   ...
}
```

当使用多重子类型时，将类 MyDBObject 嵌套在 StorableShape 里，那么成员 DBObject._name 会隐藏掉成员 StorableShape._name。所以，如果在 MyDBObject 内部的代码只是简单地使用_name，那么它将引用到 DBObject._name：

```
import std.array;

class StorableShape : Shape {
   private class MyDBObject : DBObject {
      override void saveState() {
         // 修改外层的 Shape 对象 Shape._name
         this.outer._name = "A";
         // 修改基对象 DBObject._name
         _name = "B";
         // 仅将指针清理干净
         assert(super._name == "B");
      }
   }
   private MyDBObject _store;
   alias _store this;
   this() {
      _store = new MyDBObject;
   }
   ...
}
```

6.14　参数化的类和接口

有时会需要使用静态已知的实体来参数化类或接口。假设要定义一个栈接口。此接口就应该使用存储在栈里的类型来参数化，这样我们才能避免出现重复（StackInt、StackDouble、StackWidget……）。在 D 语言里，可以像下面这样定义一个参数化的接口：

```
interface Stack(T) {
   @property bool empty();
   @property ref T top();
   void push(T value);
```

自在读书 www.zizidiary.com

```
    void pop();
}
```

语法 (T) 会引入一个类型参数到 Stack。在 Stack 内部，使用 T 就像使用任何类型一样。在客户端代码里使用 Stack 时，需要指定一个参数，就像在调用只有一个形参的函数时，必须传递一个实参一样。通过使用二元操作符“!”可以传递实参，如：

```
unittest {
    alias Stack!(int) StackOfInt;
    alias Stack!int SameAsAbove;
    ...
}
```

如果只有一个实参（Stack 就是这种情况），那么它两边的括号可以省略。

逻辑上的下一步是在类里实现此接口。理想情况下，这个接口实现也应该是泛型的（不针对特定的元素类型特例化）。因此，我们定义一个参数化的类 StackImpl，其接受的类型参数为 T，并将这个参数传递到 Stack，然后在这个实现里使用它。现在我们将一个数组用做后端来实际实现一个栈。

```
class StackImpl(T) : Stack!T {
    private T[] _store;
    @property bool empty() {
        return _store.empty;
    }
    @property ref T top() {
        assert(!empty);
        return _store.back;
    }
    void push(T value) {
        _store ~= value;
    }
    void pop() {
        assert(!empty);
        _store.popBack();
    }
}
```

使用 StackImpl 与实现它一样好玩：

```
unittest {
    auto stack = new StackImpl!int;
    assert(stack.empty);
    stack.push(3);
    assert(stack.top == 3);
    stack.push(5);
    assert(stack.top == 5);
    stack.pop();
    assert(stack.top == 3);
    stack.pop();
    assert(stack.empty);
}
```

一旦实例化一个参数化类，它便成为一个类。因此，StackImpl!int 跟其他类一样，也是一个类。这个特殊类之所以能实现 Stack!int，是因为“甜饼模具”StackImpl(T) 会将其定义里的所有 T 都替换成 int。

自在读书 www.zizidiary.com

再论异构转换

既然我们已聊过如何从参数化类型产生出实际的类型，那么现在就来仔细看看实例化机制。首先，在 5.3 节中我们讨论了在类型参数化函数的环境下的异构转换概念（相对于同构转换）。现在回顾一下，在同构转换里，语言基础设施针对所有值（例如，全部都是 Object）采用了一种公共格式，然后在引擎作用下将泛型（类型参数化的）代码调整到公共格式。这个调整可能包括来回强制类型转换以及将某些类型“装箱”（boxing）的操作，使它们遵从公共值格式；然后在用户代码请求它们时，将它们“拆箱”（unboxing）。整个过程是类型安全的，并且完全透明。Java 针对其参数化类型都使用了异构转换。

在同构方法里，对于所有的类型，所有 StackImpl 都会共享相同的方法实现代码。更为重要的是，在类型这一级它们都是没有区别的——动态类型 StackImpl!int 和 StackImpl!double 都是相同的。转换器基本上只会为所有的 Stack!T 定义一个接口，而为所有的 StackImpl!T 定义一个类。它们叫做“擦除”（erased）类型，因为它们会擦除所有与 T 相关的信息。然后，转换器会巧妙地将使用带有各种 T 的 Stack 和 StackImpl 的代码进行替换，让它们只使用那些被擦除的类型。不管客户端代码使用的 Stack 和 StackImpl 带有何种类型，关于这些类型的静态信息都不会被保存。这种信息是用于静态类型检查的，它会很快被遗忘——最好是擦除掉。基于部分信息会出现丢失这样一个简单的事实，可以判定这个过程是有问题的。有一个简单的例子，即你不能重载函数 Stack!int 和 Stack!double——因为它们是同一种类型。其中还有更深层的可靠性问题，它们都需要进一步进行讨论，并且在一些研究文献[14，1，49]中有部分被提及。

异构转换器（如 C++的模板机制）的方法有所不同。对一个异构转换器来说，Stack 并不是类型，它只是一种创建类型的方式（补充说明：类型是值的模板，而参数化类型是类型模板）。对于 Stack!int、Stack!string 或者在应用程序中需要的任何类型，每一个的实例化都将生成一个截然不同的类型。不管你正使用的 Stack 具有什么样的类型，异构转换器在使用它来替换 T 时，都会通过复制和粘贴 Stack 函数体的方式生成全部的类型。这种方法容易产生更多的代码，但它也更强大，因为它保持了静态类型信息的完整性。此外，由于异构转换是针对每种情况都生成特化的代码，因此它有可能产生更快的代码。

D 语言完全使用异构转换，这就意味着 Stack!int 和 Stack!double 是截然不同的接口，并且 StackImpl!int 与 StackImpl!double 也是截然不同的类型。除了源自相同的参数化类型外，这些类型是不相关的。当然，你也可以将它们关联起来，例如通过让所有 Stack 实例都继承一个公共的接口。由于 StackImpl 会为每种实例化类型都生成一组方法，所以会存在很多重复的二进制代码。这一点非常让人不快，因为生成的代码事实上经常可能是完全相同的。聪明的编译器应该会将所有相同的函数合并成一个（在撰写本书时，官方的 D 编译器还无法做到这一点，但是这种合并在更为成熟的 C++编译器里是一项经过反复验证的技术）。

自在读书 www.zizidiary.com

当然，一个类可以拥有多个类型参数，只需要在 StackImpl 的实现里加一点有意思的调整，便能得到这样的效果。我们不是将存储绑到数组上，而是将这一决定转移到 StackImpl 的外面。在数组的所有功能里，StackImpl 只使用了 empty、back、~=以及 popBack。现在我们来让容器决定 StackImpl 的实现细节：

```
class StackImpl(T, Backend) : Stack!T {
    private Backend _store;
    @property bool empty() {
        return _store.empty;
    }
    @property ref T top() {
        assert(!empty);
        return _store.back;
    }
    void push(T value) {
        _store ~= value;
    }
    void pop() {
        assert(!empty);
        _store.popBack();
    }
}
```

6.15　小结

在 D 语言里，类是实现面向对象设计的主要机制。它们总是使用引用语义并被垃圾回收。

继承是动态多态性的体现。D 语言只支持状态的单一继承，但一个类可以继承多个 interface 类型。接口没有状态，但可以定义 final 方法。

保护规则遵从操作系统的保护（目录和文件）。

所有类都继承了 Object 类——它定义在由 D 语言实现提供的 object 模块里。Object 定义了一些重要的原语，而模块 object 则定义了进行对象比较的关键函数。

一个类可以定义嵌套类——它会自动存储一个对父类的引用，而静态嵌套类则不会存储。

D 语言完全支持非虚接口（NVI）惯用法，也支持多重子类型化的半自动机制。

自在读书 www.zizidiary.com

第 7 章

其他自定义类型

许多优秀的软件都可以使用类、基础类型和函数来编写。使用类型和值来参数化的类和函数让事情变得更加简单。但很显然，类并不是抽象策略的最终类型，其中的原因有如下几点。

第一，类遵从的是引用语义，这一点可能使它们在表示大量的设计时效果很差，而且开销很大。像二维或三维坐标这样的简单实体，如果有上百万个，那么使用类来建模是件很困难的事，而且这会让设计师处于一种两难的境地：在好的抽象和合理的效率之间该做何选择呢。同样地，在线性代数里，别名（aliasing）会造成很大的麻烦。你很难说服数学家或科学领域的程序员，让他们认可：使用 a = b 对两个矩阵进行赋值后，a 和 b 都应该引用同一个实际矩阵，而不是让 a 拥有 b 的独立副本。当与 D 语言精练的抽象类型（第 4 章）相比时，即使像数组那样简单的类型也会让以类为基础的模型出现很大的开销。当然，数组可能会有所不同，但经验不断表明：提供大量用户代码无法企及的魔法类型是一项令人感到沮丧的提议，同时也是一种缺乏语言设计的表现。一个数组的有效负载为两个机器字，因此对所有数组的原语来说，分配一个类对象并使用一个附加的间接引用，意味着空间和时间都会存在更大的开销。即使像 int 这样的简单类型也无法以一种低代价且优雅的方式表示成类，哪怕我们将操作符便捷性的问题完全忽略掉也不会有改观。像 BigInt 那样的类会再次面临这种问题：a = b 所做的事情与对应的 int 赋值存在很大的差异。

第二，类拥有无限生命周期，就是说它们很难为生命周期非常有限并且供应也相当欠缺的资源（如文件句柄、图形句柄、互斥锁、套接字等）建立模型。使用类来处理这类资源会将你的注意力牢牢地吸引住，因为你必须记得使用像 close 或 dispose 这样的方法及时地释放掉那些被封装的资源。scope 语句（参见 3.13 节）能提供不少帮助，但它适合的场合是封装那种作用域完完全全限定在一种类型里的语义。

第三，类是一个相对较重且高层的抽象机制，所以它们并不适合用于表达像枚举类型或者为给定类型换名这样的抽象概念。

如果 D 语言将类作为唯一的抽象机制，那么它将不能成为一门真正的系统级编程语言。除了类，D 语言还支持很多存储形式，其中包括 struct（此值类型拥有类的绝大部分强大功能，

自在读书
www.zizidiary.com

但具有值语义，没有多态机制）、enum（轻量级列举类型以及简单的常量）、union（与类型无关的底层交错存储）以及辅助类型定义机制（如 alias）。本章将依次对它们进行查看。

7.1 结构

结构（struct）支持定义简单的、封装好的值类型。一个非常值得一提的模型是 int：一个 int 值由 4 个字节构成，同时它支持某些操作。在 int 里，没有隐藏的状态和间接引用，两个 int 总是会引用到完全不同的值[①]。struct 的宗旨就是要排除动态多态性、函数改写、继承以及无限生命周期。struct 就是发扬光大后的 int。

先回想一下，类拥有引用语义（参见 6.2 节），这意味着你总是可以通过对象的引用来操纵它；复制引用实际上只是增加了更多对同一个对象的引用，而不是复制了实际的那个对象。相比之下，struct 是值类型，这意味着它们基本上“很像 int”：名字与它们所表示的值是紧密粘在一起的，在复制结构的值时实际上复制的是整个对象，而非仅仅只有引用。

struct 的定义与 class 的定义很相似，不同之处有以下几点。

- 使用 struct 来代替 class。
- 结构不支持类继承和接口实现，因此它不能指定“: 基类型, 接口”这样的内容，并且在结构内部显然没有可引用的父级（super）。
- struct 方法不允许改写——所有方法都是最终的（可以为 struct 的方法指定 final，不过这是多此一举）。
- 不能将 synchronized 与 struct 一起使用（参见第 13 章）。
- struct 不支持定义默认构造函数 this()（这是一个需要进一步解释的问题，参见 7.1.3.1 节）。
- 在 struct 内部可以定义 postblit[②]构造函数 this(this)（参见 7.1.3.4 节）。
- 结构不支持 protected 访问指示符（即不存在派生结构）。

现在来定义一个简单的结构：

```
struct Widget {
    // 常量
    enum fudgeFactor = 0.2;
    // 共享的不可更改的值
    static immutable defaultName = "A Widget";
    // 分配给每一个 Widget 对象的状态
    string name = defaultName;
    uint width, height;
    // 静态方法
    static double howFudgy() {
        return fudgeFactor;
    }
```

① 使用别名造成的简单名字相等情况除外，关于这一点参见 7.4 节。

② 可理解为“事后块复制”，这是一种针对结构的复制构造技术。这种类型的构造函数会在复制结构时（如将结构传递给函数）被调用。　——译者注

自在读书 www.zizidiary.com

```
    // 方法
    void changeName(string another) {
      name = another;
    }
}
```

7.1.1　复制语义

在结构（struct）和类（class）之间在表面上没什么差异的地方其实却有着深层次的区别。现在再来做一个实验，我们首先要借用 6.2 节定义的类。这次我们要创建一个包含相同字段的类和结构，我们来体验一下这两种类型的复制行为：

```
class C {
    int x = 42;
    double y = 3.14;
}

struct S {
    int x = 42;
    double y = 3.14;
}

unittest {
    C c1 = new C;
    S s1;                          // 对于 S 没有 new：栈分配
    auto c2 = c1;
    auto s2 = s1;
    c2.x = 100;
    s2.x = 100;
    assert(c1.x == 100);           // c1 和 c2 都引用到同一个对象...
    assert(s1.x == 42);            // ...但 s2 是 s1 的一个真实副本
}
```

由于结构没有引用，因此需要通过使用初始化和赋值来进行绑定和重新绑定。结构值的任何名字都是与不同值相关联的。定义 auto s2 = s1;会逐个字段地复制整个结构对象。如前所述，结构对象拥有值语义，而类对象具有引用语义。图 7-1 显示的是 c2 和 s2 刚定义之后的状态。

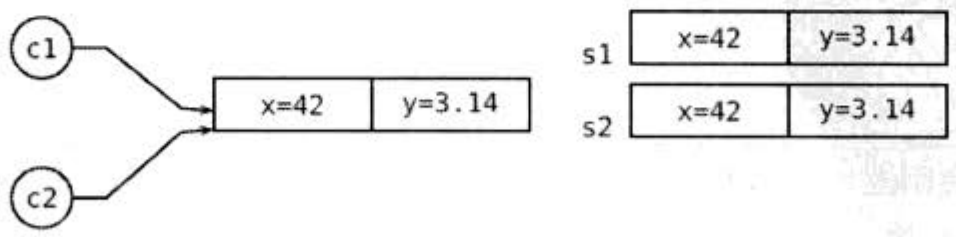


图 7-1　针对类对象 c1 计算 auto c2 = c1;与针对结构对象 s1 计算 auto s2 = s1;有着明显不同的效果，因为类的本质是引用，而结构的本质是值

c1 和 c2 拥有绑定到任何对象的能力；与之不同的是，s1 和 s2 仅仅是与已有对象不可分割的名字。两个名称无法引用同一个结构对象（除非使用别名 alias，它会定义一个简单的等效符号，参见 7.4 节），并且名字一定附着有结构——比较操作 s1 is null 是无意义的，而且它是一个编译时错误。

自在读书 www.zizidiary.com

7.1.2　将结构对象传递给函数

因为值语义的关系，结构对象会以值的方式复制到函数。

```
struct S {
    int a, b, c;
    double x, y, z;
}
void fun(S s) {              // fun 会接收一个副本
    ...
}
```

要通过引用传递结构对象，请使用 ref 参数（参见 5.2.1 节）：

```
void fun(ref S s) {          // fun 会接收一个引用
    ...
}
```

涉及 ref 时，this 会以隐藏参数 ref S 的形式传递到结构 S 的方法里。

7.1.3　结构对象的生命周期

与类对象有所不同，结构对象的生命周期是有限的。它们遵循非常精确的作用域生命周期语义，这类似于函数里的临时对象（存在于栈里）的生命周期。

想要创建一个结构对象，只需像使用函数一样使用其名字即可：

```
import std.math;

struct Test {
    double a = 0.4;
    double b;
}

unittest {
    // 将结构名当作函数一样来创建对象
    auto t = Test();
    assert(t.a == 0.4 && isnan(t.b));
}
```

调用 Test() 会创建一个结构对象，其中的所有字段会被初始化成默认值。在这个示例里，t.a 为 0.4，而 t.b 为 double.init。

也可以调用 Test(1) 和 Test(1.5,2.5)，并且它们会以声明的顺序依次初始化对象的字段。继续前一个示例：

```
unittest {
    auto t1 = Test(1);
    assert(t1.a == 1 && isnan(t1.b));
    auto t2 = Test(1.5, 2.5);
    assert(t2.a == 1.5 && t2.b == 2.5);
}
```

自在读书 www.zizidiary.com

创建结构对象的表达式 Test(‹args›)与创建类对象的表达式 new Test(‹args›)之间的语法差异起初可能有些让人不太容易接受。在创建类对象时，D 语言可以完全将关键字 new 扔掉，但保留这个 new 的目的在于提醒程序员：这里执行了一个内存分配操作(很重要的工作)。

7.1.3.1　构造函数

也可以像定义类的构造函数（参见 6.3.1 节）一样定义结构的构造函数：

```
struct Test {
    double a = 0.4;
    double b;
    this(double b) {
        this.b = b;
    }
}
unittest {
    auto t = Test(5);
}
```

只要存在一个及以上这样的构造函数，上面讨论过的所有面向字段的构造函数都会被禁用：

```
auto t1 = Test(1.1, 1.2);                    // 错误!
// 无可以匹配 Test(double, double)的构造函数
static Test t2 = { 0.0, 1.0 };               // 错误!
// 无可以匹配 Test(double, double)的构造函数
```

这里有一个很重要的例外情况，即编译器总是会定义无参数的构造函数。

```
auto t2 = Test(); // 正确，创建默认初始化的对象
```

同时，用户代码不能定义无参数的构造函数：

```
struct Test {
    double a = 0.4;
    double b;
    this() { b = 0; }                        // 错误!
    // 结构不能定义默认构造函数!
}
```

为什么会有这种限制呢？这一切都与 T.init 有关，这是每一种类型都会定义的默认值。T.init 必须是静态已知的，这一点与可执行任意代码的构造函数存在矛盾。对于类，T.init 为空引用，而非默认构造的对象。对于结构，默认构造函数在初始化一个对象时，会将它的每个字段都默认初始化。

7.1.3.2　转发构造函数

这里将 6.3.2 节的示例复制过来，并将 class 更改为 struct。

```
struct Widget {
    this(uint height) {
        this(1, height);                     // 转向其他构造函数
    }
    this(uint width, uint height) {
        this.width = width;
        this.height = height;
    }
    uint width, height;
    ...
}
```

自在读书 www.zizidiary.com

不需要进行其他修改，这段代码即可运行。与类一样，结构也支持从一个构造函数转移到另一个构造函数，并且具有相同的限制。

7.1.3.3　构造步骤

类需要考虑动态内存分配和初始化基对象（参见 6.3.3 节）的问题。对于结构，事情则要简单很多，因为内存分配步骤在构造顺序上不是隐式的。构造 T 类型结构对象的步骤如下。

（1）通过原始内存复制方式（如 memcpy），将 T.init 复制到将包含此对象的内存。

（2）如果构造函数可用，则调用它。

如果结构的某些或所有字段都是使用= void 进行初始化的，那么第一步里的工作可以省掉。但这种情况很少，因为它常会往代码里引入一些莫名奇妙的 bug（不过也有一种较为合理的情况，大家参见 6.3.3 节的 Transmogrifier 示例）。

7.1.3.4　postblit 构造函数 this(this)

假设我们要定义一个对象，在其内部拥有一个私有数组，并有限地对外公开了一个使用该数组的 API：

```
struct Widget {
   private int[] array;
   this(uint length) {
      array = new int[length];
   }
   int get(size_t offset) {
      return array[offset];
   }
   void set(size_t offset, int value) {
      array[offset] = value;
   }
}
```

这里的 Widget 定义存在一个问题：复制 Widget 对象会在副本间创建一个长距离依赖。假设：

```
unittest {
   auto w1 = Widget(10);
   auto w2 = w1;
   w1.set(5, 100);
   w2.set(5, 42);                    // 也会设置 w1.array[5]!
   assert(w1.get(5) == 100);         // 失败啦!?!
}
```

问题出在哪里？从 w1 到 w2 的复制是“浅的”，即按字段复制时，没有连带复制可能被每一个字段引用的任何内存。复制一个数组时并不会分配一个新的数组，它只会复制数组的边界（参见 4.1.4 节）。在复制之后，w1 和 w2 确实包含的是不同数组字段，但它们引用的却是同一段内存区域。这种属于值类型但却包含有间接共享引用的对象常被戏称为“类结构”（clucts）：它是具有值语义的结构与具有引用语义的类的混合物[①]。

通常，结构想要定义的是真正的值语义，即要求副本与本源是完全独立的。为此，需要像

① “clucts”这个术语是 Bartosz Milewski 创建的，即将 class 与 struct 各取部分字母拼接而成。

自在读书 www.zizidiary.com

下面那样定义一个 postblit 构造函数：

```
struct Widget {
    private int[] array;
    this(uint length) {
        array = new int[length];
    }
    // postblit 构造函数
    this(this) {
        array = array.dup;
    }
    // 同以前一样
    int get(size_t offset) { return array[offset]; }
    void set(size_t offset, int value) { array[offset] = value; }
}
```

在对象复制期间，postblit 构造函数便会发挥作用。当通过另一个相同类型的结构对象 src 初始化对象 tgt 时，编译器会执行如下几个步骤。

（1）将 src 的原始内存复制到 tgt 的内存里。

（2）对于每一个定义了 this(this) 方法的传递性字段（如字段的字段……），自底向上（内层字段优先）调用该方法。

（3）调用 tgt 的 this(this) 方法。

名字“postblit”源自“blit”，是“BLock Transfer”（块复制）的流行缩写形式，所表示的含义是原始内存复制。D 语言提供了在初始化时进行原始复制对象的功能，并在复制完成之后提供了一个钩子。在上面的示例里，postblit 构造函数利用现在已别名化的数组，并对它进行全面复制，同时确保从此处开始 Widget 的源对象和目标对象不再有什么共同的内容。在 postblit 构造函数的作用下，下面的测试现在便能顺利通过：

```
unittest {
    auto w1 = Widget(10);
    auto w2 = w1;                   // 这里会针对 w2 调用 this(this)
    w1.set(5, 100);
    w2.set(5, 42);
    assert(w1.get(5) == 100);       // 通过
}
```

不管是否会显式创建命名变量，每当进行对象复制时，postblit 构造函数调用总是会被插入。例如，将 Widget 以值的形式传递给函数时便会涉及副本创建。

```
void fun(Widget w) {                // 值传
    w.set(2, 42);
}

void gun(ref Widget w) {            // 引用传递
    w.set(2, 42);
}

unittest {
    auto w1 = Widget(10);
    w1.set(2, 100);
    fun(w1);                        // 在这里会创建一个副本
    assert(w1.get(2) == 100);       // 通过
    gun(w1);                        // 不进行复制
```

自在读书 www.zizidiary.com

```
    assert(w1.get(2) == 42);    // 通过
}
```

postblit 复制过程的第 2 步（“传递性字段”那部分）需要特别说明一下。这种行为的基本原理就是封装——结构对象的 postblit 构造函数一定会被调用，即使此结构嵌套在另一个结构对象里也一样。例如，让 Widget 成为另一个结构的成员，而这个结构又是另一个结构的成员：

```
struct Widget2 {
    Widget w1;
    int x;
}

struct Widget3 {
    Widget2 w2;
    string name;
    this(this) {
        name = name ~ " (copy)";
    }
}
```

现在，如果你想要复制这些包含 Widgets 的对象，并且如果编译器此时忘了正确地复制 Widgets 的子对象，那么情况将会变得很糟。这就是为什么在复制类型为 Widget2 的对象时，需要为 w 子对象发起对 this(this)的调用，即使 Widget2 根本就没有截断复制也需要这样做。同样地，当复制类型为 Widget3 的对象时，对于字段 w2 的字段 w1，this(this)也会再一次被调用。请仔细看看下面的代码：

```
unittest {
    Widget2 a;
    a.w1 = Widget(10);                          // 分配数据
    auto b = a;                                 // 因 b.w1 而调用 this(this)
    assert(a.w1.array !is b.w1.array);          // 通过

    Widget3 c;
    c.w2.w1 = Widget(20);
    auto d = c;                                 // d.w2.w1 而调用 this(this)
    assert(c.w2.w1.array !is d.w2.w1.array);    // 通过
}
```

总之，如果你为结构类型定义了 postblit 构造函数 this(this)，编译器便会保证每当复制此对象时，不管它是独立的还是属于另一个更大结构对象的一部分，这个 postblit 构造函数总是会被调用。

7.1.3.5　this(this)存在的原因

postblit 构造函数背后的原因是什么呢？在其他语言中并没有使用 this(this)的情况。为什么不只是将源对象传递到目标对象（C++的实现方式）呢？

```
// 这段不是 D 语言代码
struct S {
    this(S another) { ... }
// 或者
    this(ref S another) { ... }
}
```

自在读书 www.zizidiary.com

C++的应用经验已经表明：对象的过度复制是 C++程序效率低下的罪魁祸首。为减少效率的损失，C++设立了这样的情形：编译器可以忽略对复制构造函数的调用。而忽略的规则很快就变得相当复杂，并且仍然不能涵盖所有的情况，使得此问题一直悬而未决。新的 C++标准通过定义一个新类型“右值引用”解决了这些问题。这种新类型允许用户对复制的省略操作进行控制，而付出的代价就是让语言变得更加复杂。

因为有了 postblit 构建函数，D 语言省略复制的方法，因此变得很简单，并且基本上是自动实现的。首先，D 语言对象必须是可重定位的（relocatable），即与位置无关：一个对象可以通过使用原始内存移动方法进行移动，而其完整性不会受到影响。不过这个限制也意味着这些对象不能嵌入所谓的“内部指针”（internal pointer），即此对象的子部分地址。这种技术也不是必不可少的，因此 D 语言并没有为此制定过多的规则。在 D 语言里，创建带有内部指针的对象是非法的，编译器和运行时子系统都不太会主动遵守这个规则。可重定位对象给了编译器和运行时子系统（如垃圾回收器）很大的自由，从而可以让程序运行得更快，并且变得更紧凑。

由于有了可重定位对象，对象复制便变成对象移动的一个逻辑扩展：postblit 构造函数 `this(this)` 使复制等同于移动加上可选的自定义钩子（hook）。这样，用户代码就没有机会改变源对象的字段。这样做之所以很好，是因为复制不应该影响到源对象，但它要有机会解决那些不应该与源对象一起间接共享状态的字段。为了省略复制，只要可以证明副本的源对象在复制过程完成之后不会再被使用，编译器便不会插入对 `this(this)` 的调用。例如，有一个函数，它能以传值的方式返回一个 `Widget`（如上所述）对象：

```
Widget hun(uint x) {
    return Widget(x * 2);
}

unittest {
    auto w = hun(1000);
    ...
}
```

有一种省事的方法，即在 `hun` 内部简单地创建一个 `Widget` 对象，然后通过按位复制的方式将其复制到 `w`，紧跟着调用 `this(this)`。但这种做法比较浪费，因为 D 语言会假定对象总是可重定位的。既然这样，那为什么不简单地将 `hun` 创建的即将消失的临时变量移动到 `w` 里呢？没人能真正说出其中的缘由，因为在 `hun` 返回之后，临时变量都没有用处。在森林里如果一棵树倒下，是不会有人注意到它的，那么只移动“这棵树”而不去复制它也应该不会有什么问题。这里有一个相似但不相同的示例：

```
Widget iun(uint x) {
    auto result = Widget(x * 2);
    ...
    return result;
}

unittest {
    auto w = iun(1000);
    ...
}
```

自在读书 www.zizidiary.com

同样地，在这个情况里，result 在 iun 返回之后会消失，因此调用 this(this) 是没必要的。最后，还有下面这样一种奇妙的情形：

```
void jun(Widget w) {
    ...
}

unittest {
    auto w = Widget(1000);
    ... // ‹代码1›
    jun(w);
    ... // ‹代码2›
}
```

这种情况则很难摆脱对 this(this) 的调用。‹代码 2›有可能会继续使用 w，这时，将它从 unittest 移动到 jun 是不正确的[①]。

鉴于上面所有的注意项，D 语言在复制省略方面采取了以下立场。

- 所有匿名右值都会被移动，而不会被复制。当源对象是一个匿名的右值时（如上面的 hun 函数所展示的临时变量），this(this) 调用不会被插入。
- 所有在函数内部栈上分配的、然后被返回的命名临时变量都会省略调用 this(this)。
- 无法保证其他潜在省略都被发现了。

不过，有时候我们会非常想要让编译器执行一个移动。这实际上是可行的，方法就是使用在标准库模块 std.algorithm 里定义的 move 函数：

```
import std.algorithm;

void kun(Widget w) {
    ...
}

unittest {
    auto w = Widget(1000);
    ... // ‹代码1›
    // 插入对 move 的调用
    kun(move(w));
    assert(w == Widget.init); // 通过
    ... // ‹代码2›
}
```

使用 move 可以确保移动 w，以及确保会有一个空的、默认构造的 Widget 替换 w 的内容。顺便说一下，在这里名叫 Widget.init 的无状态无异常抛出默认构造函数（参见 7.1.3.1 节）正好能派上用场。如果没有这个，可能就没有办法将移动源设置为定义良好的、空的状态。

7.1.3.6　析构与解除分配

结构可以定义一个析构函数，拼写形式为~this()：

```
import std.stdio;

struct S {
```

① 此外，‹代码 1›可能会保存一个指向‹代码 2›所使用的那个 w 的指针。

自在读书 www.zizidiary.com

```
    int x = 42;
    ~this() {
        writeln("An S with payload ", x, " is going away. Bye!");
    }
}

void main() {
    writeln("Creating an object of type S.");
    {
        S object;
        writeln("Inside object's scope");
    }
    writeln("After object's scope.");
}
```

上面的程序肯定会输出如下内容：

```
Creating an object of type S.
Inside object's scope
An S with payload 42 is going away. Bye!
After object's scope.
```

任何结构对象都遵循作用域生命周期（scoped lifetime），即对象的生命周期最终会在它的作用域末尾处结束。再具体一些，即：

- 在函数里定义的非静态对象的生命周期会在当前作用域末尾结束，结束顺序是在所有比它先定义的结构对象之前；
- 定义成另外一个结构成员的对象，其生命周期会在该封闭对象的生命周期结束后立即结束；
- 在模块作用域里定义对象，其生命周期是无限的。如果想调用该对象的析构函数，那么必须在模块的析构函数里完成（参见 11.3 节）；
- 定义成类内部成员的对象，其生命周期会在该封闭对象的内存被回收时结束。

D 语言保证会在结构对象生命周期的末尾自动调用~this。如果想自动执行某些如关闭文件和释放任何敏感资源那样的操作，那么这一点是非常方便的。

使用 this(this) 的那个复制源会遵循其正常生命周期的规则，但那个挪动了内存且被省略了的复制源，其析构函数则会被正常调用。

结构对象的解除分配操作在概念上是在析构之后立即进行的。

7.1.3.7　拆毁步骤

默认情况下，结构对象的销毁顺序与创建顺序完全相反。例如，在某个作用域里最后定义的那个结构对象会先被销毁。

```
import std.conv, std.stdio;

struct S {
    private string name;
    this(string name) {
        writeln(name, " checking in.");
        this.name = name;
    }
    ~this() {
```

自在读书 www.zizidiary.com

```
        writeln(name, " checking out.");
    }
}

void main() {
    auto obj1 = S("first object");
    foreach (i; 0 .. 3) {
        auto obj = S(text("object ", i));
    }
    auto obj2 = S("last object");
}
```

上面程序的输出为：

```
first object checking in.
object 0 checking in.
object 0 checking out.
object 1 checking in.
object 1 checking out.
object 2 checking in.
object 2 checking out.
last object checking in.
last object checking out.
first object checking out.
```

正如所料，第一个被创建的对象最后一个被销毁。在循环里的对象，其创建和销毁则是以每次穿过循环为周期，分别发生在进入和退出那个控制语句作用域的时候。

通过 clear(obj) 可以显式地调用结构对象的析构函数。在 6.3.5 节我们认识了 clear 函数。在那里，clear 被用于将类对象重置为默认构造状态。对于结构对象，clear 具有类似的行为：它会调用析构函数，然后将 .init 的内容复制到对象。其结果也是一个有效的对象，只是它未保留任何有意义的状态。

7.1.4　静态构造函数和析构函数

一个结构可以定义任意数量的静态构造函数和析构函数。这个特性与 6.3.6 节介绍过的关于类的同名特性几乎是相同的。

```
import std.stdio;

struct A {
    static ~this() {
        writeln("First static destructor");
    }
    ...
    static this() {
        writeln("First static constructor");
    }
    ...
    static this() {
        writeln("Second static constructor");
    }
    ...
    static ~this() {
        writeln("Second static destructor");
    }
```

自在读书 www.zizidiary.com

```
}

void main() {
   writeln("This is main speaking");
}
```

在静态构造器和静态析构函数之间不需要进行配对。在执行 main 函数之前，运行时支持会沉闷地按所有静态构造函数的定义顺序依次执行它们。在 main 函数终止之后，运行时支持又会沉闷地按所有静态析构函数的定义顺序反向执行它们。上面这个程序会输出如下内容：

```
First static constructor
Second static constructor
This is main speaking
Second static destructor
First static destructor
```

在同一个模块里，静态构造函数和析构函数的执行顺序是定义良好的（如前所述），但跨模块时情况就有所不同了。对于跨模块的静态构造函数和析构函数，它们的执行顺序定义在 6.3.6 节。

7.1.5　方法

结构可以定义成员函数，也就是方法。由于结构没有继承，也不支持改写，因此结构的方法与普通函数差别不大。

对于结构类型 S，非静态方法会通过引用接收一个隐藏参数 this（相当于 ref S 参数）。在方法内部，名称查找过程与类一样：参数会掩盖同名的成员名称，而成员名称会掩盖同名的模块级名称。

```
void fun(int x) {
   assert(x != 0);
}

// 例证名字查找规则
struct S {
   int x = 1;
   static int y = 324;

   void fun(int x) {
      assert(x == 0);                    // 获取参数 x
      assert(this.x == 1);               // 获取成员 x
   }

   void gun() {
      fun(0);                            // 调用方法 fun
      .fun(1);                           // 调用模块级的 fun
   }

   // 单元测试可以是结构的成员
   unittest {
      S obj;
      obj.gun();
      assert(y == 324);                  // "成员"单元测试可以看到静态数据
   }
}
```

自在读书 www.zizidiary.com

上面的示例表明单元测试也可以定义在结构内部。这种成员单元测试并没有什么特别之处，它只是能非常方便地插入到每个方法定义的后面。在成员单元测试里的代码拥有与常规静态方法相同的名称可见性。例如，上面的单元测试便没有在静态成员 y 的前缀里加上 S，就像静态方法不用进行添加一样。

有几个特殊的方法这里有必要进一步调查一下。它们是用于“=”的赋值操作符 opAssign，用于“==”和“!=”的相等操作符 opEquals，以及用于“<”、“<=”、“>=”、“>”的顺序操作符 opCmp。这些内容其实是第 12 章的主题，因为那一章关注的是操作符重载。但是这些操作符很特殊，因为编译器可能会自动生成它们，并让其具有某种特定的行为。

7.1.5.1　赋值操作符

默认情况下，对于下面这段代码，其中的赋值操作是按成员一个接一个进行的。

```
struct Widget { ... } // 与 7.1.3.4 节所定义的一样
Widget w1, w2;
...
w1 = w2;
```

这可能给 7.1.3.4 节讨论过的 Widget 带来麻烦。回想一下，Widget 拥有一个私有的 int[] 成员，它被认为在每一个 Widget 对象里都是不同的。将 w2 逐个字段地赋与 w1，这个操作也会将 w2.array 赋与 w1.array——这是一种简单的数组边界赋值，实际上不会复制数组的内容。这一点需要进行修正，因为我们想要的是为源 Widget 里的那个数组创建一个副本，接着再将这个副本赋与目标 Widget。

用户代码可以通过定义 opAssign 方法拦截赋值操作。从本质上讲，如果 lhs 定义了具有兼容性签名的 opAssign（否则，如果 lhs 和 rhs 拥有相同的类型，那么它会执行默认的逐个字段的赋值操作），那么赋值运算 lhs = rhs 会被翻译成 lhs.opAssign(rhs)。现在来定义 Widget.opAssign：

```
struct Widget {
   private int[] array;
   ... // this(uint)、this(this)等
   ref Widget opAssign(ref Widget rhs) {
      array = rhs.array.dup;
      return this;
   }
}
```

这个赋值操作会返回一个对 this 的引用，其目的在于允许链式赋值，如 w1 = w2 = w3。编译器会将这种链式赋值表达式重写为 w1.opAssign(w2.opAssign(w3))。

这里还剩下一个问题。假设有下面的赋值操作：

```
Widget w;
...
w = Widget(50); // 出错!
// 不能将类型为 Widget 的右值绑定到 ref Widget!
```

这里的问题在于定义的 opAssign 所期望的是 ref Widget，即一个类型为 Widget 的左值。为接受除左值以外还会来自右值的赋值，Widget 必须定义两个赋值操作符：

自在读书 www.zizidiary.com

```
import std.algorithm;

struct Widget {
   private int[] array;
   ... // this(uint)、this(this)等
   ref Widget opAssign(ref Widget rhs) {
      array = rhs.array.dup;
      return this;
   }
   ref Widget opAssign(Widget rhs) {
      swap(array, rhs.array);
      return this;
   }
}
```

在这个接受右值的版本里不再有.dup。这是为什么呢？这个右值（其后面是 array）实际上属于第二个 opAssign：它是进入函数前被复制的，并且会在函数返回之前被销毁。也就是说，完全没必要复制 rhs.array，因为不会有人再使用它。将 rhs.array 与 this.array 进行交换就足够了。当 opAssign 返回时，rhs 会随着 this 的旧 array 一起消失，而与 this 待一起的是 rhs 的旧 array——完美的状态保护。

现在我们可以完全删除 opAssign 的第一个重载：那个以传值形式接收 rhs 的重载会负责所有的事情（左值会被自动转换成右值）。但保留左值版本则能实现一项很有用的优化：opAssign 不会复制源对象，它会检查当前的 array 是否拥有足以容纳新内容的空间；在这种情况下，只用一个改写便已足够。

```
// 在Widget内部...
ref Widget opAssign(ref Widget rhs) {
   if (array.length < rhs.array.length) {
      array = rhs.array.dup;
   } else {
      // 调整长度
      array.length = rhs.array.length;
      // 复制数组内容（参见4.1.7节）
      array[] = rhs.array[];
   }
   return this;
}
```

7.1.5.2　比较结构相等

结构类型的对象可以立即使用“==”和“!=”进行相等相比。比较操作是逐个成员进行的，并且如果被比较的对象里至少有两个对应的成员出现不等的情况，则会得到 false 值；否则，得到 true 值。

```
struct Point {
   int x, y;
}
unittest {
   Point a, b;
   assert(a == b);
   a.x = 1;
   assert(a != b);
}
```

自在读书 www.zizidiary.com

为定义一个自定义比较例程，需要定义 opEquals 方法：

```
import std.math, std.stdio;

struct Point {
    float x = 0, y = 0;
    // 增加的
    bool opEquals(ref const Point rhs) const {
        // 进行近似比较
        return approxEqual(x, rhs.x) && approxEqual(y, rhs.y);
    }
}

unittest {
    Point a, b;
    assert(a == b);
    a.x = 1e-8;
    assert(a == b);
    a.y = 1e-1;
    assert(a != b);
}
```

与类的 opEquals（参见 6.8.3 节）相比，结构的 opEquals 更为简单一些，因为它不需要担心存在继承时出现的正确性问题。编译器会将结构对象的比较简单地重写成对 opEquals 的调用。当然，结构还必须定义一个有意义的 opEquals：满足自反性、对称性和传递性。需要指出的是，虽然 Point.opEquals 看起来很不错，但它的传递性测试会失败。更好一些的测试是比较两个已将其最重要部分截断后的 Points，这种测试可以让传递变得更容易。

如果结构包含的成员定义了 opEquals，但是它自身并没有定义 opEquals，那么对于定义了 opEquals 的成员，比较操作仍然会调用它们的 opEquals。继续上面的 Point 示例：

```
struct Rectangle {
    Point leftBottom, rightTop;
}

unittest {
    Rectangle a, b;
    assert(a == b);
    a.leftBottom.x = 1e-8;
    assert(a == b);
    a.rightTop.y = 5;
    assert(a != b);
}
```

假定有两个 Rectangles 类型的对象 a 和 b，计算 a == b 等同于：

```
a.leftBottom == b.leftBottom && a.rightTop == b.rightTop
```

它也可以反过来写成下面的样子：

```
a.leftBottom.opEquals(b.leftBottom) &&
    a.rightTop.opEquals(b.rightTop)
```

这个示例还表明：比较是以字段声明的顺序（即 leftBottom 在 rightTop 之前）进行的；而如果某两个字段因&&的“短路”计算而不相等，则会提前停止。

自在读书 www.zizidiary.com

7.1.6　静态成员

结构可以定义静态数据和静态成员函数。除了具有受作用域限制的可见性和特有的访问规则（参见 7.1.7 节）外，静态成员函数与常规函数拥有相同的规则要求。静态成员函数没有隐藏的 this 参数，也没有涉及其他的特殊机制。

同样地，除了被定义它们的结构施以的可见性和访问外，静态数据成员拥有与模块级的全局数据一样的规则（参见 5.2.4 节）。

```
import std.stdio;

struct Point {
   private int x, y;

   private static string formatSpec = "(%s %s)\n";

   static void setFormatSpec(string newSpec) {
   ... // 检查格式说明的正确性
      formatSpec = newSpec;
   }

   void print() {
      writef(formatSpec, x, y);
   }
}

void main() {
   auto pt1 = Point(1, 2);
   pt1.print();
   // 通过加入前缀 Point 或 pt1 来调用静态成员
   Point.setFormatSpec("[%s, %s]\n");
   auto pt2 = Point(5, 3);
   // 新的格式说明将影响所有的 Point 对象
   pt1.print();
   pt2.print();
}
```

上面的程序预计会输出下面的内容：

```
(1 2)
[1, 2]
[5, 3]
```

7.1.7　访问指示符

结构类型遵从的访问指示符有 private（参见 6.7.1 节）、package（参见 6.7.2 节）、public（参见 6.7.4 节）以及 export（参见 6.7.5 节），其方式与类类型的访问指示符一致。对结构来说，protected 没有意义，因为结构没有继承。

想要了解全部的内容，请查阅一下相应的各节。在这里，我们只简要看一下每一种访问指示符的含义：

自在读书 www.zizidiary.com

```
struct S {
   private int a; // 在本文件内可访问，S 的方法也可以访问
   package int b; // 在文件所在目录内可访问
   public int c;  // 本应用程序内可访问
   export int d;  // 本应用程序外可以访问
                  //     （应用程序所在的地方）
}
```

同样地，尽管 export 在语法上可以出现在任何支持设置访问指示符的地方，但其语义是由编译器实现决定的。

7.1.8　嵌套结构和类

通常，在结构内部定义其他结构或类能带来很大的方便。例如，树容器可以选择对外公开具有简单搜索接口的结构外壳，但在内部却使用多态性来定义树的节点。

```
struct Tree {
private:
   class Node {
      int value;
      abstract Node left();
      abstract Node right();
   }
   class NonLeaf : Node {
      Node _left, _right;
      override Node left() { return _left; }
      override Node right() { return _right; }
   }
   class Leaf : Node {
      override Node left() { return null; }
      override Node right() { return null; }
   }
   // 数据
   Node root;
public:
   void add(int value) { ... }
   bool search(int value) { ... }
}
```

类似地，结构也可以嵌套在其他结构内部：

```
struct Widget {
private:
   struct Field {
      string name;
      uint x, y;
   }
   Field[] fields;
public:
   ...
}
```

最后一点，结构也可以嵌套在类的内部：

```
class Window {
   struct Info {
      string name;
      Window parent;
```

自在读书 www.zizidiary.com

```
        Window[] children;
    }
    Info getInfo();
    ...
}
```

与类嵌套类不一样，在结构内部嵌套的结构和嵌套的类都不包含任何的隐藏成员 outer——不会生成任何特殊代码。嵌套这些类型的主要设计目的是要加强实际需要的访问控制。

7.1.9　在函数内部嵌套结构

回想一下 6.11.1 节的内容，嵌套类会获得一些独特的特性。嵌套类能够访问封闭函数的参数和局部变量。如果返回一个嵌套类对象，那么编译器甚至还会创建一个动态分配的函数框架，以便让函数的局部变量和参数在其结束后依然能够存活。

出于一致性的考虑，D 语言也为嵌套在函数内部的结构提供了类似的便利。嵌套结构可以访问封闭函数的参数和局部变量：

```
void fun(int a) {
    int b;
    struct Local {
        size_t c;
        int sum() {
            // 访问参数、变量以及 Local 自己的成员
            return a + b + c;
        }
    }
    Local obj;
    int x = obj.sum();
    assert(Local.sizeof == 2 * size_t.sizeof);
}
unittest {
    fun(5);
}
```

嵌套结构嵌入有神奇的“帧指针”。这种指针允许这些结构访问外部的值，如上面示例里的 a 和 b。因为有了这个额外状态，Local 对象的大小就不再是大家所想的 4，而是 8（在 32 位机），因为其中算上了帧指针。如果想让定义的嵌套结构没有这个包袱，只需在 Local 的定义里（即在 struct 的前面）加上 static，这样 Local 便成为一个常规的结构，从而阻止它去访问 a 和 b。

与嵌套类相比，嵌套结构除了避免某种无端的限制外，毫无用处。嵌套结构对象不能从函数里返回，这是因为调用者无法访问到它们的类型。如果某个设计使用了复杂的嵌套结构，那么它在无形中便会生成复杂的函数，最好是在第一时间就该避开的这一点。

7.1.10　结构子类型化与@disable 属性

尽管结构不具有继承和多态性的特性，但它们仍然支持 alias this 特性（它首先在 6.13

自在读书 www.zizidiary.com

节引入)。通过 alias this，可以让结构子类型化到其他任何类型。例如，我们定义一个名叫 Final 的简单类型，并让其行为看上去像类引用——只是你绝不能重新绑定它！下面是一个使用 Final 的例子：

```
import std.stdio;

class Widget {
    void print() {
        writeln("Hi, I'm a Widget. Well, that's about it.");
    }
}
unittest {
    auto a = Final!Widget(new Widget);
    a.print();              // 正确，只会输出 a
    auto b = a;             // 正确，a 和 b 都绑定到了同一个 Widget
    a = b;                  // 错误！
                            // opAssign(Final!Widget)被禁用了！
    a = new Widget;         // 错误！
                            // 不能给 get()返回的右值进行赋值！
}
```

Final 的目的是要成为一个特殊的类引用，它首先会绑定到某个对象，然后便绝不能与之分离。这种“忠实的”引用在许多设计里很有用处。

第一步是摆脱赋值。这里的问题在于赋值操作符如果不存在则会自动生成，所以 Final 必须友好地向编译器说明不必那样做。为达到这种效果，可使用@disable 属性：

```
struct Final(T) {
    // 禁止赋值
    @disable void opAssign(Final);
    ...
}
```

你可以使用@disable 来删除另外生成的函数，如比较函数。

到目前为止，一切都很顺利。为实现 Final!T，我们需要确保该类型能使用 alias this 将 T 子类型化，同时不会提供左值。错误的设计会像下面这个样子：

```
// 错误的设计
struct Final(T) {
    private T payload;
    this(T bindTo) {
        payload = bindTo;
    }
    // 禁止赋值
    @disable void opAssign(Final);
    // 子类 T
    alias payload this;
}
```

Final 在成员 payload 里保存它的值，并且会在构造函数中初始化它。它实际上也禁止了赋值操作，即声明了 opAssign 但没有定义它。这样，客户端代码在试图给类型为 Final!T 的对象进行赋值时，便无法进行访问（因为 private 的关系）或遇到一个连接时错误。

Final 的错误之处在于它引入了 alias payload this。这里有一个单元测试，它完成了一些本不该发生的事情：

自在读书 www.zizidiary.com

```
class A {
    int value = 42;
    this(int x) { value = x; }
}
unittest {
    auto v = Final!A(new A(42));
    void sneaky(ref A ra) {
        ra = new A(4242);
    }
    sneaky(v); // 嗯...
    assert(v.value == 4242); // 能通过？！？
}
```

alias payload this 的工作方式很简单。只要类型为 Final!T 的值 obj 用于对其类型来说是非法的环境时，编译器便会将 obj 重写为 obj.payload。换句话说，它会使 obj.payload 成为 obj 的别名，这正是此特性的名称和语法。但是 obj.payload 直接访问的是 obj 的某个字段，因此它是一个左值。这个左值所绑定的是 sneaky 的 ref 参数，因此也允许 sneaky 直接改写 v 的字段。

为了修正此问题，需将该对象别名成一个右值。这样便能获得完整的功能，但存储于 payload 里的引用是不可触及的。绑定到右值非常容易，具体方法就是使用@property，它会以值传形式返回 payload：

```
struct Final(T) {
    private T payload;
    this(T bindTo) {
        payload = bindTo;
    }
    // 通过未定义的 opAssign 来禁止赋值
    private void opAssign(Final);
    // 子类 T，但不允许重新绑定
    @property T get() { return payload; }
    alias get this;
}
```

其中的关键点是：get 返回的是一个 T，而非 ref T，因此 payload 不能被重新绑定。当然，所引用的那个 payload 是可以被修改的（为达到阻止修改的目的，可以引用到第 8 章所讲到的 const 和 immutable）。Final 的宗旨现在得到了落实。首先，对于任何类类型 T，Final!T 的行为都与 T 一样。其次，一旦将 Final!T 绑定构造函数里的某个对象，便不能将它重新绑定到任何其他对象。也就是说，这个单元测试无法再进行编译，因为现在调用 sneaky(v)是非法的：一个类型为 A 的右值（通过 v.get 可以隐式从 v 获得）不能绑定到 sneaky 的所需要的 ref A，因为这种做法很有问题。

还有一个小问题（真的很小）需要解决。每当像 Final 那样的类型使用 alias 获得 this 时，大家就需要特别小心：Final 自己的符号会掩盖掉被别名类型所定义的同名符号。例如，我们在使用 Final!Widget 时，Widget 自身也定义了一个名为 get 的特性：

```
class Widget {
    private int x;
    @property int get() { return x; }
}
unittest {
```

自在读书 www.zizidiary.com

```
    auto w = Final!Widget(new Widget);
    auto x = w.get; // 获得 Final 里的 Widget
                    // 而不是 Widget 里的 int
}
```

为了避免这种冲突，我们需要使用一种命名协定，而坚实的协定即是以透明特性的名义简单地使用类型的名字：

```
struct Final(T) {
    private T Final_payload;
    this(T bindTo) {
        Final_payload = bindTo;
    }
    // 禁止赋值
    @disable void opAssign(Final);
    // 子类 T，但不允许重新绑定
    @property T Final_get() { return Final_payload; }
    alias Final_get this;
}
```

利用这种协定，出现不可预料的冲突的风险就大大降低了。当然，有时也会很想要拦截某些方法，那么便可以将它们放置在拦截器里。

7.1.11 字段布局与对齐

结构对象里的字段是如何排列的呢？D 语言对于结构是非常保守的——它将结构的内容以结构定义里指定的相同顺序进行布局，但仍然可以随意地在字段之间插入“填补”(padding)。假设：

```
struct A {
    char a;
    int b;
    char c;
}
```

如果编译器完全依照 A 所规定的大小来摆放各个字段，那么 b 就应该位于对象地址加 1 的位置上（一个 char 字符恰好占用 1 个字节）。这是错误的，因为当代大多数的计算机系统都是以 4 个或 8 个字节的增量方式获取数据的，而且限制它们只能从那些地址分别为 4 或 8 的倍数的地方获取数据。现在假定有一个类型为 A 的对象，它正位于一个“好”地址上，如恰好是 8 的倍数。那么，b 的地址显然会处于一个“坏”位置上。为获得 b，处理器需要完成的动作相当繁重——需要将 b 值的各个字节大小的零散部分组配起来，以获得完整的值。更为雪上加霜的是，由于依赖于编译器和底层硬件架构，整个操作都可能陷在响应“错位内存访问”的内核陷阱里，而这也有它自己的（有时会更大）开销[28]。这显然是不划算的——这些额外的动作很容易将访问速度降低好几个数量级。

出于这个原因，现今的编译器都使用“填补”来排列结构。编译器会往这些对象里插入一些额外的字节，从而确保所有字段都正好处在有利的偏移量上。然后，在大小为字的倍数的地址上分配这些对象，便可以保证快速访问到它们的所有成员。图 7-2 展示了 A 在填补后的布局情况。

自在读书 www.zizidiary.com

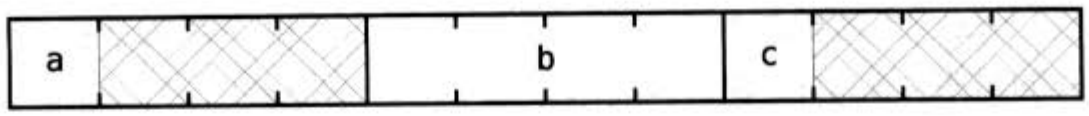


图 7-2　类型为 A 的对象填补后的布局情况。阴影区域是用于适当对齐而插入的填补。编译器往对象里插入了两个窟窿，同时还增加了 6 个字节的宽松空间，而对象的总大小则增加了 50%

最终的布局拥有相当多的填补内容（阴影字节）。对于类，编译器可以随意地重新排列各个字段；但对于结构，如果内存消耗很重要的话，则需要特别注意数据的布局。就前面那个示例定义的结构而言，较好的字段排序方法是首先放置 int，然后紧跟两个 char 字符，它们再加上 2 个字节的填补便正好符合 64 位。

一个对象的每个字段都有一个编译时已知的相对此对象起始地址的偏移量。在给定的应用程序里，这个偏移量对所有相同类型的对象来说都是相同的（在编译时它们会变化，而在运行时不会变化）。用户可以访问此偏移量，访问方法是使用类或结构里的所有字段都会隐式定义的.offsetof 特性：

```
import std.stdio;

struct A {
   char a;
   int b;
   char c;
}

void main() {
   A x;
   writefln("%s %s %s", x.a.offsetof, x.b.offsetof, x.c.offsetof);
}
```

这个参考实现会输出“0 4 8”，它们揭示的布局与图 7-2 所示的一致。令人尴尬的是，为能访问到关于它的某种静态信息，你需要创建一个类型为 A 的对象，但像 A.a.offsetof 这样的语法却是无法编译的。不过，这里可以使用一个技巧，即使用 A.init.a.offsetof，它能以编译时常量的形式获得与每个成员相关联的偏移量。

```
import std.stdio;

struct A {
   char a;
   int b;
   char c;
}

void main() {
   // 无对象情况下访问字段偏移
   writefln("%s %s %s", A.init.a.offsetof,
      A.init.b.offsetof, A.init.c.offsetof);
}
```

D 语言保证所有的填补字节始终为 0。

align 属性

如果想要改写编译器的对齐选择（它会影响到插入的填补），那么可以在字段声明里使用

自在读书 www.zizidiary.com

一个对齐修饰符 align。当需要与指定了特定对齐特性的某块硬件或某种二进制协议打交道时，可能会有这样的需求。align 说明符看起来像下面这个样子：

```
class A {
   char a;
   align(1) int b;
   char c;
}
```

有了上面的说明，A 的字段在布局时其间是没有空隙的。在对象的末尾则可能存在空隙。align 的参数是字段的最大对齐，因此实际的对齐不可能超出字段类型的自然对齐。为获得 T 类型的自然对齐，请使用编译器定义的属性 T.alignof。因此，假如在上面例子里，将 b 的 align(1)替换成 align(200)，那么得到的有效对齐为 4，与 int.alignof 一样。

你也可以将 align 与结构的整个定义一起使用：

```
align(1) struct A {
   char a;
   int b;
   char c;
}
```

当用于结构指定时，align 会将默认对齐设置成一个特定值。在结构定义内部，你可以使用单个的 align 属性改写默认对齐属性。对于类型为 T 的字段，如果只指定了 align 而没有提供具体数字，那么它会被当成 align(T.alignof)，换句话说，它会将此字段的对齐重置为自然对齐。

align 不允许与指针和引用一起使用。垃圾回收器假设所有引用和指针都是以 size_t 大小对齐的。编译器不会强制要求这个限制，因为有时指针和引用并不是垃圾回收的。因此，这个定义是相当危险的，因为在编译时这个问题不会出现警告：

```
struct Node {
   short value;
   align(2) Node* next; // 应避免
}
```

如果上面的代码执行 obj.next = new Node 这样的赋值（如使用垃圾回收引用来填充 obj.next），那么必然会产生混乱：垃圾回收器会扫描到这个错误对齐的引用，内存被回收，接着 obj.next 变成了一个悬空引用。

7.2　联合

D 语言也接受 C 语言风格的联合（union），不过应该说明一下的是，最好能少这样做，并且要特别小心。

联合与结构很像，所有成员都起始于同一个地址。因此，它们的存储是重叠的，也就是说你要负责读取你所写入的那一种类型。在任何给定的时候，都只有一个联合的成员值是有效的。

```
union IntOrFloat {
   int _int;
   float _float;
}
```

自在读书 www.zizidiary.com

```
unittest {
    IntOrFloat iof;
    iof._int = 5;
    // 只读取 iof._int，而不会读取 iof._float
    assert(iof._int == 5);
    iof._float = 5.5;
    // 只读取 iof._float，而不会读取 iof._int
    assert(iof._float == 5.5);
}
```

因为 int 和 float 的大小相同（4 字节），所以在 IntOrFloat 里它们将完全重叠。不过，它们的具体布局是未指定的。例如，_int 和_float 都可能使用不同的字节顺序：_int 的最高有效字节可能在最低地址，而_float 的最高有效字节（它包含有符号和大部分指数）可能是在最高地址。

联合不会进行标记，即联合对象本身不会包含“标记”，而这是区分某个成员是否完好的方式。正确使用联合的责任就直接落在了用户的肩上，在使用联合构建某些更大型抽象概念时，这一点让人很不舒服。

如果在定义时没有进行初始化，联合对象会初始化成第一个字段的.init 值，因此，在默认构造之后第一个成员是可读的。为使用其他不同于.init 的值来初始化第一个字段，则需要在大括号里指定需要的初始化值：

```
unittest {
    IntOrFloat iof = { 5 };
    assert(iof._int == 5);
}
```

静态联合对象可以使用下面的语法来初始化不同的字段：

```
unittest {
    static IntOrFloat iof = { _float : 5 };
    assert(iof._float == 5);
}
```

老实说，联合的很多用途实际上都是用于读取跟写入时完全不同类型的值，其目的在于实现某些与系统相关的表示操作。正因为这一点，编译器才不会对那些甚至是可探测的联合错误用法产生报怨。例如，下面的代码在 Intel 32 位机器上能通过编译，甚至断言也能通过：

```
unittest {
    IntOrFloat iof;
    iof._float = 1;
    assert(iof._int == 0x3F80_0000);
}
```

联合可以定义成员函数，也可以定义所有结构通常可定义的成员，但不包括构造函数和析构函数。

联合最常见（最好这样做）的用法是用作结构内部的匿名成员，如下所示：

```
import std.exception;

struct TaggedUnion {
    enum Tag { _tvoid, _tint, _tdouble, _tstring, _tarray }
    private Tag _tag;
    private union {
```

自在读书 www.zizidiary.com

```
        int _int;
        double _double;
        string _string;
        TaggedUnion[] _array;
    }

public:
    void opAssign(int v) {
        _int = v;
        _tag = Tag._tint;
    }
    int getInt() {
        enforce(_tag == Tag._tint);
        return _int;
    }
    ...
}

unittest {
    TaggedUnion a;
    a = 4;
    assert(a.getInt() == 4);
}
```

关于 enum 的细节在本章即将讨论到，参见 7.3 节。

上面的示例是绝对经典的关于联合的用法，即在定义所谓的“可辨识联合”（discriminated union），又名标记联合（tagged union）或代数类型时，可以将它作为一种辅助工具。TaggedUnion 在安全盒里封装了一个不安全的联合，这个安全盒会跟踪最后分配的那个类型。在初始化时，Tag 的值为 Tag._tvoid，这意味着该对象实际上是未初始化的。当给这个联合赋值时，opAssign 会恰如其分地介入进来并设置此对象的类型。为完成此实现，还需要定义 opAssign(double)、opAssign(string) 和 opAssign(TaggedUnion[])，以及 getXxx() 函数。

这里的联合成员是匿名的，这意味着它既是一个类型定义，也是一个成员定义。匿名联合会以结构内部普通成员的方式进行分配，而其成员在结构内部都是直接可见的（如 TaggedUnion 的方法所体现的）。通常可以同时定义匿名结构和匿名联合，并将它们嵌套成你想要的样子。

最后，大家应该知道联合并不像看起来那么邪恶。在你与编译器之间，使用联合代替 cast 进行类型转换通常是一种很好的沟通方式。指针和整数的联合可以向编译器表明它应该保留着，并且不要回收这个指针。如果你以整数形式存储这个指针，并且偶尔将它强制转换回指针的类型，那么最后的结果是未定义的，因为垃圾回收器可能已将与这个暗指针关联的内存回收。

7.3 枚举值

实践证明，只接受几个离散值的类型是非常有用的——事实上，它相当有用，以至于 Java 在多年英勇地试图使用惯用法来模仿它们之后，最终还是将枚举类型添加到核心语言里[8]。良好的枚举类型（enum）是不容易定义的——在 C++和（尤其是）C 语言里，枚举都带有一丝古

自在读书 www.zizidiary.com

怪的气息。D 语言则尽力运用“事后诸葛”的优势定义了一个简单实用的枚举设施。

从基础开始吧。枚举最简单的用法是“让我列举一些符号值吧”，同时不会将它们与新的类型关联起来：

```
enum
   mega = 1024 * 1024,
   pi = 3.14,
   euler = 2.72,
   greet = "Hello";
```

枚举的类型推断过程与 auto 一样。所以，在上面的示例里，pi 和 euler 都为 double 类型，而 greet 的类型为 string。如果要定义多个特定类型的枚举，那么可以紧跟关键字 enum 的后面指定一个类型：

```
enum float verySmall = 0.0001, veryBig = 10000;
enum dstring wideMsg = "Wide load";
```

所有枚举值都是常量。在本质上，使用它们时就相当于使用它们所代表的字面量。更特别的是，所支持的操作都是相同的——例如，你不能提取 pi 的地址，这跟你不能提取 3.14 的地址是一样的：

```
auto x = pi;                        // 正确，x 的类型为 double
auto y = pi * euler;                // 正确，y 的类型为 double
euler = 2.73;                       // 错误！不能修改枚举值！
void fun(ref double x) {
   ...
}
fun(pi);                            // 错误！不能获取 3.14 的地址！
```

如上所示，枚举值的类型并不仅限于 int，还可以是 double 和 string。那什么样的类型才可用于枚举呢？答案很简单：所有基础类型和 struet 类型都可以用于枚举。对枚举值的初始化，只有以下两点要求。

- 初始化值在编译过程中必须是可计算的。
- 初始化值的类型必须允许复制，即不为@disable this(this)（参见 7.1.3.4 节）。

第一个要求确保了枚举值不会依赖于任何运行时参数。第二个要求确保了可以实际创建值的副本——每当使用枚举的时候，都会创建一个副本。

不能定义类类型的枚举值，因为类对象总是使用 new 来创建的（除了无什么意义的 null 值），并且在编译过程中 new 是不可计算的。如果这个限制在未来被取消或者放宽，也不足为奇。

现在来创建和使用一个结构类型的枚举：

```
struct Color {
   ubyte r, g, b;
}

enum
   red = Color(255, 0, 0),
   green = Color(0, 255, 0),
   blue = Color(0, 0, 255);
```

例如，不管什么时候使用 green，该代码的行为就像是将 Color(0, 255, 0)粘贴过来

自在读书 www.zizidiary.com

将符号名称替换掉一样。

7.3.1　枚举类型

一个枚举值集合可以拥有一个命名类型：

```
enum OddWord { acini, alembicated, prolegomena, aprosexia }
```

一旦将名字当做一个枚举集合，那么这些枚举值就必须是相同的类型。全部都必须拥有同一种类型，因为用户可能随即定义和使用这种类型的值。例如：

```
OddWord w;
assert(w == OddWord.acini);     // 默认初始值为此集合里的第一个值：acini
w = OddWord.aprosexia;          // 总是使用类型名来限定值名
                                //（顺便提一下，它并不是你可能认为的那个）
int x = w;                      // OddWord 可转换至 int，但反向转换则不行
assert(x == 3);                 // 值被数字化成 0，1，2，...
```

命名枚举的类型自动推导为 int。你也可以像上面这样分配一种不同的类型：

```
enum OddWord : byte { acini, alembicated, prolegomena, aprosexia }
```

对于这个新的定义（byte 被称为 OddWord 的基类型），这些枚举符号仍然具有相同的值，只是存储不同而已。也可以将类型设置为 double 或者 real，并且归属于这些符号的值仍然为 0、1 等。但如果让 OddWord 的基类型为某个非数字类型，如字符串，那么必须为所有值指定初始化值，因为编译器没有这种类型的自然连续值。

现在回到数字枚举。如果为任何枚举值分配一个特定值，那么被编译器用来给符号赋值的那个内部步长计数器会被重置。例如：

```
enum E { a, b = 2, c, d = -1, e, f }
assert(E.c == 3);
assert(E.e == 0);
```

如果两个枚举符号拥有相同的值（如 E.a 和 E.e），也不会出现冲突。实际上，你也可能在无意识的情况下创建出相等的值，因为浮点类型总能让粗心的人惊讶不已：

```
enum F : float { a = 1E30, b, c, d }
assert(F.a == F.d);             // 通过
```

上面例举的问题在于能被浮动数精确表示的最大整数为 16_777_216。在超过它之后，则会使用相同的浮点数来表示这些整数。

7.3.2　枚举特性

每一个枚举类型 E 都定义有 3 个特性：E.init 等于 E 定义的第一个值，E.min 是 E 定义的最小值，而 E.max 是 E 定义的最大值。仅当 E 的基类型支持编译期间的“<”比较时，最后两个值才会被定义。

自在读书 www.zizidiary.com

在枚举内部，你可以随意地定义自己的 min、max 和 init 值，但不推荐这样做，因为泛型代码通常依赖于这种拥有特定语义的值。

有一个大家都常会问的问题：是否能够获得枚举值的名字呢？实际上是可以的，并且很容易。但使用的机制并未内建，而是依赖于编译时的反射功能。对于某个枚举类型 Enum，编译时反射的工作方式就是暴露一个编译时常量--traits(allMembers, Enum)，此常量会以字符串元组的形式包含 Enum 的所有成员。因为字符串在运行时和编译时都能被处理，这种方法具有很大的灵活性。例如，把期望降低一点，我们只想要编写一个函数 toString，让它返回与某个枚举值相对应的字符串。这个函数需要使用 enum 类型来参数化。

```
string toString(E)(E value) if (is(E == enum)) {
    foreach (s; __traits(allMembers, E)) {
        if (value == mixin("E." ~ s)) return s;
    }
    return null;
}

enum OddWord { acini, alembicated, prolegomena, aprosexia }

void main() {
    auto w = OddWord.alembicated;
    assert(toString(w) == "alembicated");
}
```

上面有一个还未介绍过的内容 mixin("E." ~ s)，这是一个 mixin 表达式。mixin 表达式会接受一个编译时已知的字符串，并将它当做一个处于当前上下文的普通表达式，然后单纯地进行计算。在这个示例里，我们根据枚举 E 的名字、成员选择器“.”以及通过 s 迭代到的枚举值的名字，建立了这样一个表达式。在这个示例里，s 相继会变成"acini"、"alembicated"……"aprosexia"这样一些值。由此，构造出的字符串分别为"E.acini"等，它们会被 mixin 计算得到与这些符号相对应的实际值。只要传入的 value 等于其中的某一个值，这个函数便会正常返回。在传入非法的 value 时，toString 会抛出异常。为让事情变得简单些，我们选择了只返回一个空字符串。

上面展现的 toString 函数已经实现在标准库模块 std.conv 里，该模块主要用于处理常见的转换操作。名字可能有点不一样——你必须写成 to!string(w)，而非 toString(w)，这主要是出于灵活性的考虑；你也可以调用 to!dstring(w) 或者 to!byte(w) 等。同样是这个模块，还定义了可将一个字符串转换成枚举的反向函数，例如，to!OddWord("acini") 会返回 OddWord.acini。

7.4　别名

在好几个场合我们都遇到了 size_t（一个无符号整数类型，它很大，且能够容纳下任何大小的对象）。D 语言并没有定义 size_t 类型。此类型会转变成 uint 或者 ulong，具体是哪一种则取决于主机地址空间（分别对应 32 位与 64 位）。

如果打开 D 语言编译器安装目录里的文件 object.d（或 object.di），你便能找到一个

自在读书 www.zizidiary.com

类似下面这个样子的声明：

```
alias typeof(int.sizeof) size_t;
```

属性.sizeof 会测量类型（本例里的类型为 int）的字节大小。可以使用其他任何类型来取代上面的 int，其实重点并不在这个类型身上，而在于可以使用 typeof 获得的类型大小。在 32 位架构上，编译器会使用 uint 来测量对象大小；而在 64 位架构，则使用 ulong。因此，这里的别名（alias）语句会将 size_t 引入为 uint 或 ulong 的同义词。

别名声明的一般语法比上面显示的内容还要简单：

```
alias ‹已有符号› ‹新符号›;
```

‹已有符号›可以是任何有名字的内容。它可以是类型、变量和模块——只要它拥有符号名字，就可以被别名化。例如：

```
import std.stdio;

void fun(int) {}
void fun(string) {}
int var;
enum E { e }
struct S { int x; }
S s;

unittest {
    alias object. Object    Root;          // 所有类的根
    alias std               phobos;        // 包名
    alias std.stdio         io;            // 模块名
    alias var               sameAsVar;     // 变量
    alias E                 MyEnum;        // 枚举类型
    alias E.e               myEnumValue;   // 类型的值
    alias fun               gun;           // 重载函数
    alias S.x               field;         // 结构的字段
    alias s.x               sfield;        // 对象的字段
}
```

别名的应用规则很简单：只要别名符号有意义的地方都可以使用别名。编译器的做法是：从概念上将别名名字重写成被别名的符号。甚至是错误信息或被调试的程序都可以“透视到”别名，并显示原始符号，这个看起来有些让人惊讶。举个实例，在一些错误消息或调试符号里，你可能会看到 immutable(char)[]，而非 string，但具体显示哪一个则是编译器具体实现的事。

别名可以将之前的已别名内容再一次别名掉，进而形成“双重别名”。例如：

```
alias int Int;
alias Int MyInt;
```

这里没什么特别之处，仅仅是一些常规规则而已：在 MyInt 的定义处，Int 被替换成了它所别名的那个符号，即 int。

别名常用来缩短复杂的符号链，或者将源自不同模块的重载函数（参见 5.5.2 节）结合在一起。

别名的另一个常见用法是参数化的结构和类。例如：

```
// 定义一个容器类
class Container(T) {
    alias T ElementType;
    ...
```

自在读书 www.zizidiary.com

```
}

unittest {
    Container!int container;
    Container!int.ElementType element;
    ...
}
```

在这里，如果 Container 没有暴露 alias ElementType，那么外界就不会有任何合理的方式访问到与 Container 的形参 T 相绑定的那个实参。T 仅在 Container 内部可见，对外界来说则是不可见的——Container!int.T 无法编译通过。

最后，将别名与 static if 结合能发挥很大的作用。例如：

```
// 这个是 object.di
// 在两个不同的指针类型之间进行定义
static if (size_t.sizeof == 4) {
    alias int ptrdiff_t;
} else {
    alias long ptrdiff_t;
}
// 使用 ptrdiff_t...
```

ptrdiff_t 的别名声明会将其绑定到不同的类型，具体是哪种类型则取决于选择的是 static if 的哪一条分支。如果无法进行这样的绑定，那么需要这种类型的代码就必须留在 static if 的两条分支里。

7.5　使用模板实现参数化作用域

到目前为止介绍过的可让编译时参数化（类似于 C++的模板或 Java 和 C#的泛型）变得更容易的实体，包括函数（参见 5.3 节）、参数化类（参见 6.14 节）以及参数化结构，它们都遵循与参数化类相同的规则。但有时候会想要执行一些编译时类型操作，它们不会造成函数、结构或类的定义。这样的实体（广泛用于 C++）会根据静态已知的布尔条件选择一个或另一个类型。不会定义新的类型，也不会调用函数，只有一个用于这两种现有类型的别名。

有时需要编译时参数化不会定义出新类型或函数，对于这种情况，D 语言定义了参数化作用域。这种参数化作用域可以像下面这样引入：

```
template Select(bool cond, T1, T2) {
    ...
}
```

上面内容实际上就是针对刚才讨论过的编译时选择机制的骨架。很快我们便会实现它，不过在此之前，先彻底专注一下这个声明。模板（template）声明会引入一个作用，它带有名字（在本示例中为 Select）和编译时参数（在本示例中为一个布尔类型和两个类型）。模板声明可以出现在模块级、类定义里、结构定义里以及另一个模板声明里，但不能出现在函数定义里。

在参数化作用域的主体内部，允许接受任何正常的声明，并且可以使用参数名称。从外面往里看，作用域内的任何声明都可以通过在其前缀加上这个作用域名称和一个小圆点“.”进行

自在读书 www.zizidiary.com

访问，如 Select!(true, int, double).foo。事实上，在完成 Select 的定义之后，便可以立即使用它：

```
template Select(bool cond, T1, T2) {
   static if (cond) {
      alias T1 Type;
   } else {
      alias T2 Type;
   }
}

unittest {
   alias Select!(false, int, string).Type MyType;
   static assert(is(MyType == string));
}
```

请注意，我们也可以使用结构或类实现同样的效果。请别忘了，在这些类型内部都可以定义别名，而这个别名可以使用平常的小圆点“.”语法进行访问：

```
struct Select2(bool cond, T1, T2) { // 或者 class
   static if (cond) {
      alias T1 Type;
   } else {
      alias T2 Type;
   }
}

unittest {
   alias Select2!(false, int, string).Type MyType;
   static assert(is(MyType == string));
}
```

可以说这是一个不太讨人喜欢的解决方案。想象一下 Select2 的文档：“请不要创建类型为 Select2 对象！之所以定义它完全是因为其内部的那个别名！”利用特化机制来定义参数化的作用域，能让意图变得更清晰，而且不会造成混淆和滥用。

模板作用域不仅可以引入别名，实际上还可以引入任何声明。我们来定义另一个有用的模板，这次这个模板会产生一个布尔值，它可以用来辨别给定类型是否为一个任何宽度的字符串。

```
template isSomeString(T) {
   enum bool value = is(T : const(char[]))
      || is(T : const(wchar[])) || is(T : const(dchar[]));
}

unittest {
   // 非字符串
   static assert(!isSomeString!(int).value);
   static assert(!isSomeString!(byte[]).value);
   // 字符串
   static assert(isSomeString!(char[]).value);
   static assert(isSomeString!(dchar[]).value);
   static assert(isSomeString!(string).value);
   static assert(isSomeString!(wstring).value);
   static assert(isSomeString!(dstring).value);
   static assert(isSomeString!(char[4]).value);
}
```

参数化的作用域可以是递归的。举例来说，下面是一种阶乘实现：

自在读书 www.zizidiary.com

```
template factorial(uint n) {
   static if (n <= 1)
      enum ulong value = 1;
   else
      enum ulong value = factorial!(n - 1).value * n;
}
```

尽管上面的 factorial 功能很完善，但在这里它不是最好的。当想在编译期间进行计算时，你可能会想到使用编译时计算（在 5.12 节有描述）。与前面的 factorial 模板不一样，factorial 函数更为灵活，因为它可以在编译时和运行时进行计算。模板工具最好是用于类型处理，如 Select 和 isSomeString 所采用的方式。

齐名模板

模板可以定义任意数量的符号，但正如上面的示例所示，多数时候它都恰好只定义了一个。一个模板通常被定义来只完成一件事情，并以一个独立符号的形式暴露其“劳动成果”，如 Select 里的 Type 或者 isSomeString 里的 value。

需要记住和提醒的是，实例化后面的符号会变得有些格格不入。通常情况下，人们很容易忘添加.Type，并且也会疑惑 Select!(cond, A, B)为什么会产生一条令人费解的错误信息。

D 语言在这里借助了一条简单规则来缓解问题，这条规则就是著名的“齐名模板技巧”：如果某个模板定义了一个与模板本身同名的符号，那么随后模板名字的所有使用都会自动将该符号添加到所有使用此模板的地方。例如：

```
template isNumeric(T) {
   enum bool isNumeric = is(T : long) || is(T : real);
}

unittest {
  static assert(isNumeric!(int));
  static assert(!isNumeric!(char[]));
}
```

现在只要有代码使用 isNumeric!(T)，编译器便会自动将它重写成与 isNumeric!(T).isNumeric，从而让用户省去将冗余符号添加到模板名里的麻烦。

使用齐名技巧的模板在其内部也可以定义其他名字，但它们从外面是不可访问的。这是因为编译器在字查找过程中很早便进行了重写。访问这些符号的唯一方法就是在模板自身内部进行访问。例如：

```
template isNumeric(T) {
   enum bool test1 = is(T : long);
   enum bool test2 = is(T : real);
   enum bool isNumeric = test1 || test2;
}

unittest {
  static assert(isNumeric!(int).test1); // 出错!
     // bool 类型未定义名叫 test1 的特性!
}
```

自在读书 www.zizidiary.com

由齐名规则引起的错误消息为：在执行其他操作之前，编译器会将 isNumeric!(int)扩展成 isNumeric!(int).isNumeric。接着，用户代码会试着获取 isNumeric!(int).isNumeric.test1，这个相当于获取 test1 成员的 bool 值，因此出现错误消息。长话短说，当且仅当想要模板恰好只暴露一个符号时使用齐名模板。这种情况是很平常的，因此齐名模板非常流行和方便。

7.6 使用 mixin 模板注入代码

有些设计需要往多个类实现添加样板代码（如数据定义和方法定义）。典型的例子包括序列化支持、观察者模式[27]和窗口系统事件传递。

继承可用于这样的任务，但单继承实现模型对于一个给定的类想要保存多个源样板是不可能的。有时拥有这样一种机制也是很不错的：简单地将一些预定义的代码转储到某个类里，就像手写的一样。

在这里 mixin（混入）模板正好能派上用场。值得注意的是，截至今天，这个特性尚属实验性的。在将来 D 语言的修订中，有可能会使用一种更为通用的 AST 宏工具取代 mixin 模板。

mixin 模板可以像刚刚讨论过参数化作用域（模板）那样进行定义。例如，有一个 mixin 模板，它会引入一个变量、一个获取器和一个设置器，它看起来会像下面这个样子：

```
mixin template InjectX() {
    private int x;
    int getX() { return x; }
    void setX(int y) {
        ... // 检查
        x = y;
    }
}
```

一旦定义之后，mixin 模板便可以插入到多个地方：

```
// 在模块作用域内注入
mixin InjectX;

class A {
    // 注入一个类
    mixin InjectX;
    ...
}

void fun() {
    // 注入一个函数
    mixin InjectX;
    setX(10);
    assert(getX() == 10);
}
```

上面的代码现在会在模块级、在 A 类里以及在 fun 里定义那个变量和两个相关的函数，完全跟手工将 InjectX 的主体粘贴过来一样。特别是，A 的后代还可以改写 getX 和 setX，就像是 A 自己定义的一样。没有出现令人不快的重复性复制和粘贴。

当然，接下来的逻辑步骤是假定 InjectX 不接受编译时参数，但不排除有人可能会接

自在读书 www.zizidiary.com

受——而确实存在这样的情况：

```
mixin template InjectX(T) {
   private T x;
   T getX() { return x; }
   void setX(T y) {
      ... // 检查
      x = y;
   }
}
```

现在 InjectX 的使用方法需要像下面那样传递参数：

```
mixin InjectX!int;
mixin InjectX!double;
```

这个实际上会给我们带来歧义——假如你拥有了上面的两个实例，那么你现在使用 getX 会出现什么情况呢？有两个函数都具有这个名字，因此显然这里存在一个歧义性问题。为解决它，D 语言允许你在 mixin 实例化时引入作用域名称：

```
mixin InjectX!int MyInt;
mixin InjectX!double MyDouble;
```

借助这些定义，现在使用常规作用域解析明确地访问由 mixin 引入的两个成员：

```
MyInt.setX(5);
assert(MyInt.getX() == 5);
MyDouble.setX(5.5);
assert(MyDouble.getX() == 5.5);
```

所以 mixin 模板跟复制和粘贴几乎一模一样，你可以进行多次复制和粘贴，并且指定要引用的是哪一个实例。

mixin 里的符号查找

mixin 模板和常规模板（参见 7.5 节）的最大差别，以及最有可能引起混淆的地方，是查找（lookup）。

模板完全是模块化的：模板里的代码会在模板的定义位置查找符号。这是一个很值得拥有的特性，因为这意味着你可以通过分析一个模板的定义来理解它。

相比之下，mixin 模板是在实例化位置进行符号查找，这意味着你需要理解使用 mixin 模板的环境，并想象出其行为会是什么样子。

为了说明这种差别，请看下面的示例。它在定义位置和实例化位置两个地方使用了相同的名字提供符号：

```
import std.stdio;

string lookMeUp = "Found at module level";

template TestT() {
   string get() { return lookMeUp; }
}

mixin template TestM() {
```

自在读书 www.zizidiary.com

```
    string get() { return lookMeUp; }
}

void main() {
    string lookMeUp = "Found at function level";
    alias TestT!() asTemplate;
    mixin TestM!() asMixin;
    writeln(asTemplate.get());
    writeln(asMixin.get());
}
```

输出结果为：

```
Found at module level
Found at function level
```

mixin 模板倾向于获得局部符号，这让它们具有了一定的表达力，但也使它们难以理解。这种行为让 mixin 模板的适用性也受到很多限制，在使用这种特殊的工具时一定要三思而行。

7.7 小结与参考

不是所有抽象概念需求都可以令人满意地使用类（class）来解决，特别是对于一些细颗粒的对象、限定作用域资源以及值类型。结构（struct）填补了这一空白。特别是，构造函数和析构函数允许限定作用域资源类型的简单定义。

联合（union）是一种底层特性，它允许你让编译器在已知重叠的存储区域里存储各种类型。

枚举（enum）就是简单的自定义离散值。枚举可被赋与一个新类型，而这会加强对该类型内定义的值的类型检查。

别名（alias）是一种非常有用的将一个符号与另一个符号关联在一起的方式。通常被别名的符号要么很长、很复杂，要么在嵌套实体里进行计算，并且一定要暴露成一个简单的名字。

使用模板（template）实现的参数化作用域对于定义如“类型自省”（type introspection）[①]和“类型特征”（type traits）这样的编译时计算是非常有用的。齐名模板允许以一种非常方便的封装形式提供抽象概念。

参数化作用域也可以以实验性的 mixin 模板形式进行实现，其行为表现很像简单的宏。一种完整的 AST 宏特性可能会取代 mixin 模板。

① 类型自省指的是程序在运行时能够推断对象类型或特性的一种能力。参考 http://en.wikipedia.org/wiki/Type_intospection。——译者注

自在读书 www.zizidiary.com

第 8 章

类型修饰符

在一门语言里，类型修饰符（type qualifier）能捕获到关于类型的重要断言。这种断言对于程序员和编译器都非常有用，它们很难使用惯例、常规的子类型化（参见 6.4.2 节）或者类型参数化（参见 6.14 节）来进行捕捉。

关于类型修饰符的规范例子是 const（它由 C 语言引入并被 C++语言改进）。当它附着于类型 T 时，它会表达出这样的断言：T 值可被初始化和读取，但绝不能写入。编译器会确保这个断言得到遵守。const 修饰符在模块划分方面非常有用，因为它给调用者带去了关于函数的保证。例如，下面的签名会向 printf 的使用者承诺：printf 不会更改 format 参数里的字符。

```
// C 语言标准函数
int printf(const char * format, ...);
```

还有，这种保证对于大规模开发也是很有用的，因为它能减少由非模块化变更所造成的依赖。这种保证可以采用惯例的方式进行定义和维护，但这种惯例比较笨拙并且难以保持同步。

D 语言定义了 3 种类型修饰符。

- const（常量）表示的是上下文不变性。一个由 const 修饰的类型的值不能被直接更改。然而，在程序里的其他实体可能会写访问这个数据，比如 printf 的调用者可能对 format 进行写访问，但 printf 自身却不能这样做。
- immutable（不可变的）表示的是绝对的、上下文无关的不变性。一个由 immutable 修饰的类型的值在初始化之后，任何情况下都不能被这个程序里的任何代码更改。这是一个比 const 更强的保证。
- shared（共享的）表示的是跨线程共享的值。

这些修饰符是互为补充的。const 和 immutable 对于大规模开发很重要。immutable 支持函数式编程，而 const 让面向对象和函数式的代码接口变得更容易。immutable 和 shared 是适合并发性的工具。在第 13 章我们会对并发性进行讨论，并深入解释 shared。在这里，我们主要关注一下 const 和 immutable。

自在读书
www.zizidiary.com

8.1 immutable 修饰符

一个 immutable 值是一成不变的：在它被初始化后，便可以认为它被永久地烙在存储它的那块内存里。在整个程序的执行过程中，它永远不会改变。

修饰后的类型可拼写成：‹修饰符›(T)，这里的‹修饰符›为 immutable、const 和 shared 三者当中的某一个。例如，像下面这样定义一个 immutable 整数：

```
immutable(int) forever = 42;
```

所有试图更改 forever 的方式都会导致编译时错误。此外，immutable(int)是一种类型，它几乎跟其他类型没什么两样（只与 int 有所差别）。例如，可以将它别名到另一个符号：

```
alias immutable(int) StableInt;
StableInt forever = 42;
```

使用 auto 定义一个 forever 的副本则会将 immutable(int)类型传播到这个副本，所以这个副本自身也为 immutable(int)。这并没什么特别之处，但它标识出了类型修饰符与 static（参见 3.2.4 节）或 ref（参见 5.2.1 节）这样的简单存储类别之间的差别。

```
unittest {
    immutable(int) forever = 42;
    auto andEver = forever;
    ++andEver; // 出错！不能更改 immutable 类型变量的值！
}
```

一个 immutable 值不需要使用编译时常量进行初始化：

```
void fun(int x) {
    immutable(int) xEntry = x;
    ...
}
```

像上面那样使用 immutable 对阅读 fun 的人来说是非常有用的。他们一眼便能看出在 fun 的整个函数体里，xEntry 是 x 的入口值。

在声明里，immutable 可以不用与某个指定的类型一起使用——它可以像 auto 那样推断出具体的类型：

```
immutable pi = 3.14, val = 42;
```

编译器会将 pi 的类型推断为 immutable(double)而将 val 的类型推断为 immutable(int)。

传递性

你可以使用 immutable 修饰任何类型。例如：

```
struct Point { int x, y; }
auto origin = immutable(Point)(0, 0);
```

自在读书 www.zizidiary.com

因为 immutable(T)对于所有类型 T 来说都是一种跟其他类型没什么差别的类型，因此 immutable(Point)(0, 0)便是一个与 Point(0, 0)一样的结构字面量。

不变性会很自然地转移到对象的成员上。要知道，除了允许给 origin 整体进行赋值外，我们会希望禁止给 origin.x 或 origin.y 进行赋值。否则，整个不可变承诺便会轻易被打破。

```
unittest {
    auto anotherOrigin = immutable(Point)(1, 1);
    origin = anotherOrigin; // 出错!
    origin.x = 1;           // 出错!
    origin.y = 1;           // 出错!
}
```

事实上，immutable 会传播到 Point 的每一个字段，并使用 immutable 来修饰该字段的类型。例如，下面的断言能够顺利通过：

```
static assert(is(typeof(origin.x) == immutable(int))); // 通过
```

事情变得越来越有意思了。现在假设有一个 struct，它包含某种间接量（indirection），如一个数组字段：

```
struct DataSample {
    int id;
    double[] payload;
}
```

很显然，immutable(DataSample)字段是不能被修改的。那么怎样才能更改 payload 的元素呢？

```
unittest {
    auto ds = immutable(DataSample)(5, [ 1.0, 2.0 ]);
    ds.payload[1] = 4.5; // ?
}
```

在这里有两种抉择，会导致不同的取舍结果。一种是让这种修饰符变浅，即假定让 DataSample 变成不可变的，同时也会让它的直属字段变成不可变的，但对通过这些字段间接访问的数据没有影响[①]。另一种选择是让不变性具有传递性，这意味着在让某个对象成为不可变的同时，也会让它所能达到的任何数据变成不可变的。D 语言选择的是第二种。

具有传递性的不变性比起不具传递性的版本更为严格。如果你定义了一个不可变的值，那么连接到这个值的整个数据网（通过引用、数组和指针）也都是不可变的。因此，定义具有传递性的不可变值比定义浅不可变值更为困难。但只有付出才会有收获。基于下面两个基本原因，D 语言选择了具有传递性的不变性。

- *函数式编程*：对于函数式风格，大家都各执一词，但是大部分人都会承认：没有副作用是一个很重要的原则。依照惯例来维持这种承诺是不太靠谱的。具有传递性的不变性给编码人员提供了机会，让他们可以使用函数式风格编写出代码片段定义良好的程序，同时也让编译器有能力证实函数式代码不会无意中去修改数据。

① 这是 C++的 const 所采用的方法。

自在读书 www.zizidiary.com

- *并发编程*：并发性是一个巨大的而且非常复杂的话题，几乎不存在牢不可破的保证和颠覆不破的真理。不可变共享具有这样的确定性：跨线程共享不可变数据是正确的、安全的和有效的。为了允许编译器证实共享数据确实是不可变的，不变性必须要具有传递性。否则，访问不可变数据块的线程很容易违反规则，跟随该数据的间接字段变成可变共享。

总之，只要你拥有一个类型为 immutable(T) 的值，那么你便会知道能从该值达到的所有内容也都是不可变的。并且，没有人能改变那个数据——不可变数据跟用烙铁焊住的数据一样牢靠。这是一个非常牢固的保证，例如，它可以让你跨线程共享不可变数据，同时什么也不用担心。

8.2 immutable 的书写

由于修饰符可以使用括号精确地选择它们所修饰的类型，并且一个被修饰的类型跟其他类型是一致的，因此将 immutable 与其他类型构造器组合在一起可以创造出相当复杂的数据结构。例如，现在来比较一下下面两种类型：

```
alias immutable(int[]) T1;
alias immutable(int)[] T2;
```

在第一个定义里，括号将整个数组类型括起来了；在第二个定义里，受影响的只有 int，而那个数组不受影响。如果使用 immutable 时没有带圆括号，那么它会应用于整个类型，因此与 T1 的等价定义可以拼成下面的样子：

```
alias immutable int[] T1;
```

T1 相当简单——它只是一个不可变的 int 数组。类型的拼写正好说明了是怎么回事。由于传递性的关系，你不能整体修改这个数组（如给它赋值），你也不能修改数组的任何元素：

```
T1 a = [ 1, 3, 5 ];
T1 b = [ 2, 4 ];
a = b;        // 出错!
a[0] = b[1]; // 出错!
```

第二种定义更为奇妙，但是如果我们在头脑里将 immutable(int) 当做是一种它自己的类型，那么它就真的很简单了。于是，immutable(int)[] 便是一个那种类型的数组，仅此而已。此数组的特性接下来也是很容易进行推断的。你也可以按整体的方式给这个数组赋值，但你不能对数组的单个元素进行赋值或修改：

```
T2 a = [ 1, 3, 5 ];
T2 b = [ 2, 4 ];
a = b;        // 正确
a[0] = b[1]; // 出错!
a ~= b;       // 正确（隐约是这个样子）
```

一开始大家可能觉得这里有些奇怪，但是往一个由不可变元素构成的数组里追加是合法的。为什么会这样？原因很简单，往数组追加并未涉及更改其内部已有的内容。如果重新分配这个数组，则可能需要进行数据复制，这是很正常的。

自在读书 www.zizidiary.com

正如前面提过的（参见 4.5 节），string 实际上是 immutable(char)[]的别名。上一章已经大量描述的 string，其许多有用特性实际上都依托于修饰符 immutable。

当 immutable 与参数化类型一起使用时，这里的书写方式同样有效。例如，假设有一个泛型类型 Container!T。于是，immutable(Container!T)会修饰整个容器，而 Container!(immutable(T))只会修饰容器里的单个元素。

8.3 immutable 参数和方法

在函数签名里，immutable 的信息很丰富。假设有一个普通函数：

```
string process(string input);
```

它其实是下面这个函数的一种简洁表示：

```
immutable(char)[] process(immutable(char)[] input);
```

process 保证它不会更改 input 里的单个字符，因此 process 的调用者能确切地知道这个字符串在调用前后是完全相同的：

```
string s1 = "blah";
string s2 = process(s1);
assert(s1 == "blah");      // 绝不会失败
```

此外，调用者还可以使用源自 process 的且永不改变的结果：没有隐藏别名，也不存在某个函数以后会更改 s2 的可能。再次强调：immutable 就意味着永不可变。

结构和类都可以定义 immutable 方法。在这种情况下，这个修饰符会应用于 this 指针，并拼写成下面的样子：

```
class A {
   int[] fun();              // 常规方法
   int[] gun() immutable;    // 只能用于 immutable 对象的调用
   immutable int[] hun();    // 同上
}
```

第三种语法看起来可能有些令人费解，因为 immutable 看起来是应用于 int[]的，而实际上它是应用于 this 指针的。如果要定义一个能返回 immutable int[]的 immutable 方法，那么可以像下面这样做，只是读起来有些拗口：

```
immutable immutable(int[]) iun();
```

这时，将 immutable 放置在后面看起来会更好一些：

```
immutable(int[]) iun() immutable;
```

允许在前面位置混用 immutable 的原因在于，要与其他如 final 或 static 那样的方法属性保持一致。尤其是，你可以将下面的内容编写成一次定义多个 immutable 方法的方式：

```
class A {
   immutable {
      int foo();
```

自在读书 www.zizidiary.com

```
        int[] bar();
        void baz();
    }
}
```

即可以将 immutable 用做标签：

```
class A {
immutable:
    int foo();
    int[] bar();
    void baz();
}
```

很显然，只有 immutable 对象才可以调用 immutable 方法：

```
class C {
    Void fun(){}
    Void gan()immutable{}
}

unittest {
    auto c1 = new C;
    auto c2 = new immutable(C);
    c1.fun(); // 正确
    c2.gun(); // 正确
              // 其他调用都不工作
}
```

8.4　immutable 构造函数

处理一个不可变对象也不是太难，但是要构建一个这种对象却是一个相当复杂的过程。这是因为在构建期间需要满足两种相互冲突的需求：一个是需要给字段赋值，另一个是需要建立它们的不变性。出于这个原因，D 语言对不可变构造函数（immutable constructor）给予了特别的照顾。

不可变构造函数进行类型检查的方式有些简单和保守。编译器只允许在构造函数内部对字段进行赋值，并且禁止读取字段（包括将 this 传递给某个方法调用）。一旦不可变构造函数终止，此对象便会被“冻结”——不能再对字段进行更改。只能以读的方式调用非静态方法，因为该方法可以访问到 this，并可以潜在地读到任何字段。（编译器不会检查这个方法是否会真正读这些字段——它会保守地假设这个方法确实读某个字段。）

这条规则更具有约束性而非必要性：唯一的要求是在某个特定的字段被读取之后不要再对它进行赋值。但是这条更具约束力的规则并不会显著地削弱表达力，反而更简单和易于理解。例如：

```
class A {
    int a;
    int[] b;
    this() immutable {
        a = 5;
        b = [ 1, 2, 3 ];
        // 不允许调用 fun()
    }
```

自在读书 www.zizidiary.com

```
    void fun() immutable {
        ...
    }
}
```

在一个不可变构造函数内部，只要调用会进入到另一个不可变构造函数，那么调用父级构造函数是没问题的。将这个调用转发至另一个不可变构造函数也是合法的。这种调用并不会破坏 immutable 的完整性。

你可能常常会发现，在初始化精心设计的不可变对象时，使用递归非常有用。例如，有一个单链表类，它通过一个数组进行初始化：

```
class List {
    private int payload;
    private List next;
    this(int[] data) immutable {
        enforce(data.length);
        payload = data[0];
        if (data.length == 1) return;
        next = new immutable(List)(data[1 .. $]);
    }
}
```

为正确地初始化列表的尾部，这个构造函数会通过一个更短的数组进行自我递归。在循环里初始化列表的做法都无法编译通过，因为遍历这个被创建的列表会涉及读取它的字段，而这是被禁止的。递归操作则优雅地解决了这个问题[①]。

8.5　immutable 相关的转换

假设有下面这段代码：

```
unittest {
    int a = 42;
    immutable(int) b = a;
    int c = b;
}
```

如果类型系统顽固不化，那么它不会接受这段代码。其中涉及两种转换——首先是从 int 到 immutable(int)，其次是从 immutable(int) 到 int。要知道，这些转换在一般情况下是不合法的。例如，如果将在上面代码里的 int 替换成 int[]，那么这两种转换都将是错误的：

```
int[] a = [ 42 ];
immutable(int[]) b = a;    // 不行!
int[] c = b;               // 不行!
```

如果允许这样的转换，那么 immutable 就无法得到尊重，因为不可变数组这时会与可变数组共享它们的内容。

不过，编译器会检测并允许可变数据与不可变数据之间的某些自动转换。具体地讲，即如

① 此解决方案由 Simon Peyton-Jones 提出。

自在读书 www.zizidiary.com

果类型 T 没有可变间接性，那么编译器便会允许 T 与 immutable(T)之间的双向转换。“没有可变间接性”的本意是被间接访问的数据是不可写的，并且被递归定义成下面的样子。

- 像 int 那样的内建值类型没有可变间接性。
- 无可变间接性类型的固定大小数组没有可变间接性。
- 引用无可变间接性类型的数组和指针没有可变间接性。
- 所有字段都没有可变间接性的结构类型没有可变间接性。

例如，下面的 S1 没有可变间接性，但 S2 有：

```
struct S1 {
    int a;
    double[3] b;
    string c;
}

struct S2 {
    int x;
    float[] y;
}
```

字段 S2.y 让 S2 具有了可变间接性，因此不允许 immutable(S2)↔S2 转换。如果允许这样的转换，那么存储在 y 里的数据便会在可变对象与不可变对象之间“过度共享”（undue sharing），这将破坏 immutable 的保证。

回到本节的第一个示例，由于 int 没有可变间接性，因此编译器允许在 int 与 immutable(int)之间来回自由地进行转换。

如果想为某个结构定义这样的转换，那么需要采取一些手动措施来引导这一过程。在提供恰当的构造函数之后，编译器便可以用它来确保代码是正确的。完成繁重转换操作最简单的办法是借助于通用转换例程 std.conv.to，它能理解所有变幻莫测的修饰符转换，并总会采取恰当的动作。

```
import std.conv;

struct S {
    private int[] a;
    // 从不可写转换成可写
    this(immutable(S) source) {
        // 将此数组复制成一个非 immutable 数组
        a = to!(int[])(source.a);
    }
    // 从可写转换成不可写
    this(S source) immutable {
        // 将此数组复制成一个不可写数组
        a = to!(immutable(int[]))(source.a);
    }
    ...
}

unittest {
    S a;
    auto b = immutable(S)(a);
    auto c = S(b);
}
```

自在读书 www.zizidiary.com

这个转换不是隐式的，但是它是可能和安全的。

8.6　const 修饰符

对 immutable 不用过多进行实验便会注意到，它太过严厉从而无法在更大范围内发挥作用。没错，只要你承诺整个程序的任何地方都不会去更改某些数据，那么 immutable 非常不错。但不变性通常是一个模块化特性：你会想要保留更改某些数据的权利，而同时又想防止其他人修改它。这些数据是不可变的，因为不可变表示的是“想看这座石雕吗？这可是你的数据。”你需要一种能表达这样一种限制的方式：“你不能更改这个数据，但其他人可以。”或者，按照 Alan Perlis 的方式：“一个人的常量是另一个人的变量。”我们现在来看看类型系统怎样才能遵守 Perlis 的格言，同时不会跌入它的反面。

有一个简单的用例是一个像会输出某些数据的 print 那样的函数。print 函数不会更改传递进来的数据，因此使用 immutable 它也能工作：

```
void print(immutable(int[]) data) { ... }
unittest {
    immutable(int[]) myData = [ 10, 20 ];
    print(myData); // 正确
}
```

堪称完美。现在假设有一个 int[]，我们刚对它进行了计算，并想要输出它。这个无法工作，因为 int[]无法转换成 immutable(int)[]——如果能进行转换，那么在可变数据与表面上不可变数据之间，便存在“过度共享”。因此，print 无法输出 int[]数据。这种限制是没道理的，因为 print 无论如何都不会更改其输入，因此它应该对不可变数据和可变数据一视同仁。

这里需要的是某种共用的“或许有可能可变的”类型，这时的 print 声明为：

```
void print(mayormaynotbemutable(int[]) data) { ... }
```

没错，只是因为 mayormaynotbemutable 有点长，所以使用术语 const 来表示它。其含义是完全一样的，即 const(T)值不会被当前的代码修改，但有可能会有其他代码修改它。这种歧义所反映出的事实是：const(T)可能源自一个 T，也可能源自一个 immutable(T)。const 的这种能力使它成为完美的修饰符，可用于修饰与常规的过程式代码进行交互的函数式代码。继续上面的示例：

```
void print(const(int[]) data) { ... }
unittest {
    immutable(int[]) myData = [ 10, 20 ];
    print(myData);           // 正确
    int[] myMutableData = [ 32, 42 ];
    print(myMutableData);    // 正确
}
```

这个示例表明，可变的数据和不可变的数据都会隐式转换成 const，这也表明了其中存在某种子类型化关系。事实上，的确如此，即 const(T)是 T 和 immutable(T)的父类型，

自在读书 www.zizidiary.com

如图 8-1 所示。

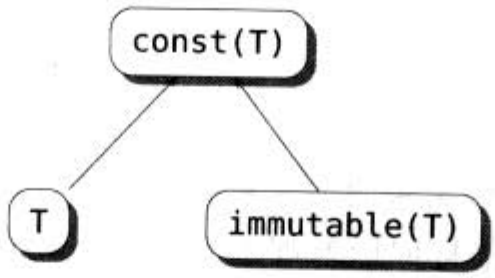


图 8-1 对于所有的类型 T，const(T) 是 T 和 immutable(T) 的父类型。这意味着处理 const(T) 值的代码也会接受可变和不变的 T 类型

const 修饰符遵循与 immutable 修饰符相同的传递性和转换规则。const 对象的构造与 immutable 对象的构造有所不同，它不会受到限制，即在 const 构造函数内部，对象被认为是可变的。

8.7 const 与 immutable 进行交互

经常会有这样的情形：一个修饰符会试图去影响一个已受另一个修饰符作用的类型。例如：

```
struct A {
   const(int[]) c;
   immutable(int[]) i;
}

unittest {
   const(A) ca;
   immutable(A) ia;
}
```

ca.i 和 ia.c 的类型是什么呢？如果盲目地应用这些修饰符，那么便会分别出现这样的类型：const(immutable(int[]))和 immutable(const(int[]))。很显然，它们都很冗长——更别提两次应用同一修饰符的 ca.c 和 ia.i 的类型！

当两个修饰符出现叠加时，D 语言会对此应用一些简单的组合规则。如果这些修饰符相同，那么它们会收缩成一个。否则，const(immutable(T))和 immutable(const(T))会收缩成 immutable(T)，因为它是信息最丰富的类型。这些规则也适用于数组内部的类型传播，例如有一个数组，其类型为 const(immutable(T)[])，那么它的元素的类型则为 immutable(T)，而非 const(immutable(T))。不过，数组自身的类型是不能简化的。

8.8 将修饰符从参数传送至结果

C 和 C++语言都定义了一个浅的 const 修饰符，它已经出现了一个烦人的问题：返回某个

自在读书 www.zizidiary.com

参数的函数必须要重复它们自己对常量和非常量数据的定义，否则日子会很不好过。经典的例子是 C 语言标准库里的函数 strchr，其 C 语言定义如下：

```
char* strchr(const char* input, int c);
```

这个函数是在“洗涮”类型：尽管 input 是一个常量值，并不期望被随意更改，再看输出，它会返回一个派生自 input 的指针，但它却将该承诺置之脑后。strchr 很容易便能产生这样的代码，即在无明显的类型转换的情况下修改不可变的数据。C++语言通过引入两种 strchr 定义解决了这个问题：

```
char* strchr(char* input, int c);
const char* strchr(const char* input, int c);
```

这两个函数所做的事情是相同的，但是却无法收缩到一起，因为C++语言并不理解这样的表述——“如果存在修饰符，那么请将它从参数传送到返回类型。”

为了解决这个问题，D 语言提供了一种“通配符”式的修饰符号，即 inout。使用 inout 之后，strchr 的声明会变成下面这个样子：

```
inout(char)* strchr(inout(char)* input, int c);
```

当然，D 语言代码偏向于使用数组而非指针。编译器知道 inout 可能会被替换成为 immutable、const 或空（后者用于会有可变输入的情形）。它会对 strchr 的函数体进行类型检查，以确保它对于所有可能的输入都可以安全地工作。

修饰符可以从方法传送到结果，例如：

```
class X { ... }
class Y {
   private Y _another;
   inout(Y) another() inout {
      enforce(_another !is null);
      return _another;
   }
}
```

another 方法对于带有任何修饰符的对象也可以工作。这个方法是可以改写的，它之所以特别，是因为 inout 可以被当做是一个泛型参数，而泛型方法通常是不可改写的。编译器能够让这个方法成为可改写的，其中的原因在于它确信对于所有修饰符底层代码都是相同的。

8.9　小结

类型修饰符用于表达类型的有用属性，这些属性很难用其他抽象机制进行表达。这里重点讨论的修饰符是 immutable，它提供了一种很强的保证，即一个 immutable 值在其整个生命周期里都绝不会被更改，且是可传递的。这是一种非常有用的特性，它确保了纯净的函数式语义，并且它对实现安全的跨线程数据共享最为重要。

自在读书 www.zizidiary.com

immutable 的长处也正是其弱点所在——它并不支持大量的将写入者与读取者责任分离的数据处理模式。const 修饰符强调的重点是上下文不变性——常量值的所有者不可修改数据，但程序的其他部分可能对这个数据拥有写权限。

最后，为避免带修饰符的函数和不带有修饰符的函数出现相同代码重复的情况，D 语言引入了通配符式的修饰符 inout。这个修饰符主要代表了 immutable、const 以及根本没有修饰符这 3 种情况。

自在读书
www.zizidiary.com

第 9 章

错误处理

错误处理（error handling）是软件工程的一个仍不太规范的领域，它关注的是如何处理那些可能存在的和预料会出现的错误情况，而且这些情况还会阻止系统的正常运作。在当今的许多语言（包括 D 语言）里，异常处理都是目前比较常用的错误处理方法，并且催生了大量指导方法、技术，甚至争议。

异常是一种语言特性，它侧重于通过附带的专用控制路径进行错误处理。当某个函数无法向其调用者返回有意义的结果时，就可能抛出一个对错误原因进行编码后的异常对象。异常抛出就是一张“脱离苦海”的卡片，它减轻了函数的正常责任。异常会跳过所有没准备处理它的调用者，并停留在一个可进一步采取紧急措施的捕捉点（catch site）。在设计良好的程序里，会有许多捕捉点，这些点比抛出点（throw site）要少。这意味着，错误处理比较集中且可重用，这一点对基于错误代码（它无处不在）的传统技术来说是很难实现的。

9.1 抛出与捕捉

D 语言使用的是一种流行的异常模型。函数可以通过 throw 语句（参见 3.11 节）引发一个异常，该语句会抛出一个类对象。为进入该对象的控制范围，代码必须使用 try 语句（参见 3.11 节），并在这条语句的 catch 子句里获取抛出的异常对象。一例抵千言，下面来看一个示例：

```
import std.stdio;

void main() {
    try {
        auto x = fun();
    } catch (Exception e) {
        writeln(e);
    }
}

int fun() {
```

自在读书
www.zizidiary.com

```
    return gun() * 2;
}

int gun() {
    throw new Exception("Going straight back to main");
}
```

函数 gun 选择不返回整数，而是抛出（throw）一个异常。在上面的示例里，这个异常即为一个 Exception 类对象。被抛出的对象可用来传输任意数量关于发生了什么事情的信息。这个异常路径让初始函数及其所有的调用者都免去了返回操作，并且会将控制传递给准备处理这个错误的调用者，即 catch 块。

在执行 throw 时，fun 会被完全跳过，因为它没准备要处理这个异常。这一点正好体现了老派错误处理方法与相对新潮的异常处理方法这两者之间的一个至关重要的区别：老派的错误处理方法必须将错误传播到所有的调用层次，而相对新潮的异常处理方法则巧妙地将控制从出错点（gun）直接转移到了特定用来处理这个错误的地方（即 main 函数里的 catch）。这种新方法采用将许多函数从四处传播错误的琐事中解脱出来的方式，实现了一种简化的、集中式的错误处理操作。因此，fun 可以无忧无虑地将 gun 和 main 之间的通信置之脑后。

但是，控制流从抛出点到捕捉点的直接传送也是异常处理的一个弱点：完全不操心实际上只是一个白日梦。在现实中，被异常遍历的函数必须要注意附加的隐藏退出点，并且要确保程序的不变性对于所有可能的控制流都满足。D 语言提供了多种牢固的机制来确保在发生异常时不变性依然有效，在本章我们将依次讨论这些内容。

9.2　类型

D 语言的基本异常层次结构（如图 9-1 所示）很简单。

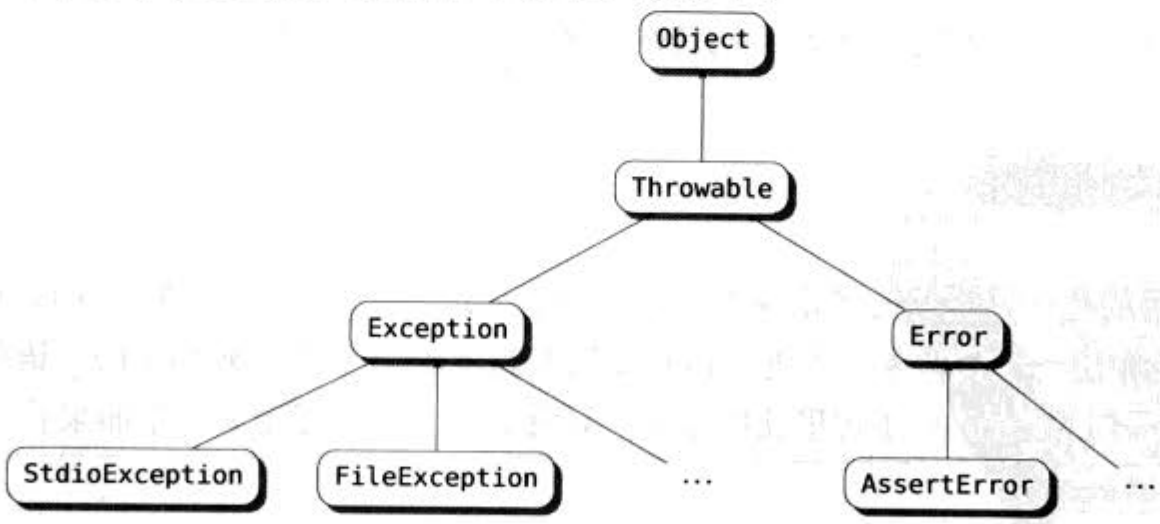


图 9-1　常规的异常继承自 Exception 类，因此可以使用 catch (Exception) 进行捕捉。Error 类直接继承自 Throwable 类。一般的代码应该只会捕捉 Exception 及其后代——而其他的类则是为了确保在程序逻辑里发现错误时可以有序地进行关闭

throw 语句抛出的内容不能是任意值，事实上只能是类对象，更具体些是继承自 Throwable 类的对象。多数情况下，代码实际上抛出的是一个继承于 Exception 的类，而 Exception 是 Throwable 的子类。这些都是正常且可恢复的异常，并且 D 语言也认可它们。继承于

自在读书 www.zizidiary.com

Throwable 的异常（Exception 除外，如 AssertError，在第 10 章会讨论它）是不可恢复的，并且在普通代码里如果真的使用它们则需要特别小心。更多关于 D 语言对不可恢复错误的保证内容和不保证内容的细节，在 9.4 节会进行讨论。

一个 try 语句可以定义多个 catch 块，例如：

```
try {
   ...
} catch (SomeException e) {
   ...
} catch (SomeOtherException e) {
   ...
}
```

异常会遵循首次匹配原则从抛出点传播到最近的那个捕捉点：只要找到一个 catch 处理器，它便会捕捉到那个抛出类或其基类，这个 catch 因此会被激活，并传递那个抛出的异常。这里有个示例，它演示了如何抛出和捕捉两个不同类型的异常：

```
import std.stdio;

class MyException : Exception {
   this(string s) { super(s); }
}

void fun(int x) {
   if (x == 1) {
      throw new MyException("");
   } else {
      throw new StdioException("");
   }
}

void main() {
   foreach (i; 1 .. 3) {
      try {
         fun(i);
      } catch (StdioException e) {
         writeln("StdioException");
      } catch (Exception e) {
         writeln("Exception");
      }
   }
}
```

上面的程序会输出下面的内容：

```
Exception
StdioException
```

第一次调用 fun 会抛出一个 MyException 对象，这个对象不会被第一个 catch 处理器匹配到，但会被第二个匹配到，因为 MyException 继承自 Exception。第二个 throw 抛出的异常会与第一个 catch 处理器直接匹配。这种首次匹配过程可能会贯穿多个函数层，如下面这个更为复杂的示例所示：

```
import std.stdio;

class MyException : Exception {
   this(string s) { super(s); }
}

void fun(int x) {
```

自在读书 www.zizidiary.com

```
    if (x == 1) {
        throw new MyException("");
    } else if (x == 2) {
        throw new StdioException("");
    } else {
        throw new Exception("");
    }
}

void funDriver(int x) {
    try {
        fun(x);
    }
    catch (MyException e) {
        writeln("MyException");
    }
}

unittest {
    foreach (i; 1 .. 4) {
        try {
            funDriver(i);
        } catch (StdioException e) {
            writeln("StdioException");
        } catch (Exception e) {
            writeln("Just an Exception");
        }
    }
}
```

上面的程序会输出如下内容，因为这些处理器在概念上会随着控制流沿调用栈向上进行尝试。

```
MyException
StdioException
Just an Exception
```

首次匹配规则有一个显而易见的结论，即如果有一个针对异常类型 E1 的 catch 处理器，紧随其后的是一个针对异常类型 E2 的 catch 处理器，而 E2 是 E1 的子类型，那么 E2 实际上是不可到达的。编译器会将这种情况标记为一个错误。例如：

```
import std.stdio;

void fun() {
    try {
        ...
    } catch (Exception e) {
        ...
    } catch (StdioException e) {
        ...     // 出错！
                // 无法到达的 catch 处理器！
    }
}
```

尽管像这样的代码可以肯定是错误的，但在跨越不同函数的动态遮掩情形下常会发生这样的事情。有些函数很轻易便能让其调用者的 catch 处理器不起作用。然而，在多数时候这并不是一个错误，而仅仅是调用栈动态变化的一种正常情况。

9.3 finally 子句

try 语句在结束时可以带上一条 finally 子句，其基本含义是“就算是天崩地裂也要执

自在读书 www.zizidiary.com

行这段代码”。不管有没有异常抛出，在 try 语句结束时 finally 子句都将被执行，结束的原因可能包括穿越（falling through）、异常抛出（throwing）、返回（returning）或者封闭循环中止（breaking）——所有你能想到的。例如：

```
import std.stdio;

string fun(int x) {
   string result;
   try {
      if (x == 1) {
         throw new Exception("some exception");
      }
      result = "didn't throw";
      return result;
   } catch (Exception e) {
      if (x == 2) throw e;
      result = "thrown and caught: " ~ e.toString;
      return result;
   } finally {
      writeln("Exiting fun");
   }
}
```

尽管 fun 的正常路径和异常路径都可能抛出异常或返回某种值，fun 总是会将 Exiting fun 输出到标准输出。

9.4　nothrow 函数和 Throwable 的特殊本质

函数可以声明为 nothrow（无异常抛出）：

```
nothrow int iDontThrow(int a, int b) {
   return a / b;
}
```

5.11.2 节已对 nothrow 函数有过讨论，现在来揭示一下其中的奥秘。nothrow 承诺函数不会抛出异常。这种函数仍然允许抛出预示着存在危险的 Throwable 类。基本上可认为 Throwable 是不可恢复的，因此编译器不会负责去“考虑”万一出现异常可能会是怎么回事，并且会在假设没有异常抛出的情况下进行代码优化。对于 nothrow 函数，编译器会对入口和出口序列进行简化，让它们不会包含用以应对可能出现的异常抛出的应急措施。

这点正好表明和强化了 Throwable 的特殊地位。Throwable 的第一条规则是不能捕捉 Throwable。如果要捕捉它，就不能指望调用结构的析构函数，也不能指望 finally 子句会被执行。这意味着系统的状态是不确定的，并且可能与在普通操作里依赖的任意数量的高层次不变量相违背。不过，D 语言保证基本类型的安全性和标准库的完整性仍然是有效的。你不能指望自己的应用程序状态也具有高层次的完整性，因为用于维持所说那个完整性的代码一点都未执行。如果你捕捉到一个 Throwable，那么可以只执行一些简单的操作。但多数时候，你需要将某条消息输出到标准错误设备或者某个日志文件中，并试着将所有可以保存的内容存储到一个单独的文件中，然后尽可能体面地退出。

自在读书 www.zizidiary.com

9.5 附属异常

在某些情形里，在另一个异常正在传播的同时，也可能会抛出异常。假设：

```
import std.conv;
class MyException : Exception {
   this(string s) { super(s); }
}
void fun() {
   try {
      throw new Exception("thrown from fun");
   } finally {
      gun(100);
   }
}
void gun(int x) {
   try {
      throw new MyException(text("thrown from gun #", x));
   } finally {
      if (x > 1) {
         gun(x - 1);
      }
   }
}
```

当调用 fun 时，会发生什么事情呢？情况有些让人抓狂。首先，fun 力图要抛出异常，但是正如前面对 finally 的“就算天崩地裂”那一节的描述，在 Exception 脱离 fun 时，gun(100) 一定会被调用。接下来，gun(100) 会抛出一个携带字符串"thrown from gun #100"的 MyException 对象。我们将第二个异常称作附属异常（collateral exception），以将它与最初抛出的那个异常区分开来，该异常被称为主异常（primary exception）。然后，gun 自身会使用一个 finally 子句产生一些额外的附属异常——更确切地说，总共增加 100 个。Machiavelli[①]本人可能也会被这样的代码吓到。

面对附属异常，编程语言可以做如下选择：

- 立即中止执行；
- 继续传播那个初始异常并将其他异常忽略掉；
- 让附属异常取代初始异常，然后继续传播；
- 继续以某种形式传播主异常和其他所有附属异常。

为保存关于发生什么事情的所有信息，最后一种方法是最全面的，但它也可能是最复杂的。例如，多个异常一起出现比只有一个异常的情况更加难以处理。

D 语言选择了一个简单有效的方法。每一个 Throwable 对象都拥有一个对附属 Throwable 对象的引用。这个附属对象可通过特性 Throwable.next 进行访问。如果没有（或者不再有）附属的 Throwable，那么 Throwable.next 会得到 null。这个实际上会建立一个单链表，它可以按这些附加错误发生的顺序完整保存所有关于它们的信息。此列表起始于那

① Machiavelli（马基雅维利）是 15 世纪意大利政治家和历史学家，他以主张统治者为达目的可以不择手段而闻名于世。——译者注

自在读书 www.zizidiary.com

个 Throwable 主异常。Throwable 的定义如下：

```
class Throwable {
   this(string s);
   override string toString();
   @property Throwable next();
}
```

主异常与附属异常之间进行显著区分，使异常抛出行为变得非常简单。在任何抛出点，要么有一个主异常在传播，要么没有。如果没有，那么被抛出的异常就会成为主的。否则，正被抛出的异常会被附加到那个起始于主异常的单链表末尾。继续上面的示例，现在来输出整个异常链：

```
unittest {
   try {
      fun();
   } catch (Throwable e) {
      writeln("Primary exception: ", typeid(e), " ", e);
      while ((e = e.next) !is null) {
         writeln("Collateral exception: ", typeid(e), " ", e);
      }
   }
}
```

上面的代码会输出如下内容：

```
Primary exception: Exception thrown from fun
Collateral exception: MyException thrown from gun #100
Collateral exception: MyException thrown from gun #99
...
Collateral exception: MyException thrown from gun #1
```

附属异常之所以是这样一种序列，是因为异常列表的添加动作是在抛出点完成的。每个抛出都会获取主异常（如果存在话），注册新异常，然后开启或继续抛出过程。

因为有了附属异常，D 语言代码便可以从析构函数内部和 scope 语句内部异常抛出。关于发生了什么事情的所有详细信息都可以在捕捉点得到。

9.6　栈展开与异常安全代码

在异常传播过程中，控制也会从最初的抛出点坚定不移地转移到匹配的 catch 处理器那里。在调用链上的所有函数都会被忽略掉（没错，几乎是所有）。作为传播异常的一部分，有一个名叫“栈展开”（stack unwinding）的过程，它会确保依次清理那些被忽略的函数。D 语言保证在异常传播过程中，下面的代码片段会被执行：

- 所有被忽略的函数里的那些栈上分配的结构对象的析构函数；
- 所有被忽略的函数的 finally 子句；
- 在异常抛出时仍生效的 scope(exit) 和 scope(failure) 语句。

在存在异常时，栈展开对于确保程序的正确性有着无可估量的辅助作用。使用异常的程序都有一个共同的问题，即泄漏资源。许多资源都需要以获取/释放的模式使用，而异常抛出会导致控制流“忘记”释放资源，并且很难用肉眼检测出来。最好的方式是将这些资源封装在 struct 类型里，它会在其析构函数里正确地将托管资源释放掉。这个话题已经在 7.1.3.6 节有所涉及，并且

自在读书 www.zizidiary.com

在模块 std.stdio 里的标准类型 File 便是这种封装的一个样例。File 会管理系统文件句柄和确保底层的句柄会在 File 对象被销毁时正确地关闭。File 对象可以被复制，并且有一个引用计数器会跟踪所有激活的副本。最后一个副本消失时便会关闭底层的文件。对于析构函数的这种习惯用法，C++程序员都很熟知和喜欢。（这个术语就是“资源分配即初始化”（Resource Acquisition Is Initialization），常缩写为 RAII。）也有其他一些语言和框架使用手动或自动的引用计数。

资源泄漏只是更大问题的一个具体实例。有时，执行/撤消模式可能无法明显地感觉到有资源与之关联。例如，当写入一个 HTML 文件时，有许多标签（如""）都必须最终使用与之对应的标记（""）进行关闭。一个非线性的控制流，包括抛出异常，可能会导致生成畸形的 HTML 文档。例如：

```
void sendHTML(Connection conn) {
    conn.send("<html>");
    ... // 发送文件的内容
    conn.send("</html>");
}
```

如果位于两个 conn.send 调用之间的代码提前终止了 sendHTML，那么结尾标签便不会被发送，因而会导致一个无效的流被发送到连接端。在 sendHTML 中间的 return 语句也可能会导致同样的问题，但 return 语句至少是在全面检查函数体时肉眼可见的。相比之下，异常抛出则可能源自 sendHTML（直接或间接）调用的任意函数，这使得代码审查员在评估 sendHTML 的正确性时相当困难和辛苦。此外，这种代码还存在比较突出的耦合问题，因为 sendHTML 的正确性依赖于大量其他函数可能潜在的异常抛出行为。

有一种解决方案是模仿 RAII（即使没有涉及资源）并定义一个结构，让它在析构函数里发送末尾标签。这个顶多是一种权宜之计。实际的需要是要保证特定代码的执行，而不是使用类型和对象将程序搞得一团糟。

另一个可能的解决方案是使用 finally 子句：

```
void sendHTML(Connection conn) {
    try {
        conn.send("<html>");
        ...
    } finally {
        conn.send("</html>");
    }
}
```

这种方法存在另一个问题：伸缩性，或者更确切地说，是对它的缺失。只要将两个或更多调用对堆积起来，finally 伸缩性不足的问题便会显现出来。例如，要发送一个正确配对的"<body>"标签。为此，需要嵌套两个 try/finally 块：

```
void sendHTML(Connection conn) {
    try {
        conn.send("<html>");
        ... // 发送头部
        try {
            conn.send("<body>");
            ... // 发送主体部分
        } finally {
            conn.send("</body>");
```

自在读书 www.zizidiary.com

```
        }
    } finally {
        conn.send("</html>");
    }
}
```

也可以只使用一个 finally 块和一个额外的状态变量实现同样的效果，其中状态变量用于跟踪这个函数进行了多远：

```
void sendHTML(Connection conn) {
    int step = 0;
    try {
        conn.send("<html>");
        ... // 发送头部
        step = 1;
        conn.send("<body>");
        ... // 发送中间部分
        step = 2;
    } finally {
        if (step > 1) conn.send("</body>");
        if (step > 0) conn.send("</html>");
    }
}
```

这种方法效果很好，不过现在需要将一大块代码单独用于状态管理，而这种做法会使此函数的意图模糊。

在这种情形下，最好是借助于 scope 语句。一个正运行的函数可以在执行到达某一点时植入 scope 语句。这样的话，无论代码片段需要逻辑配对什么内容都可以在物理上进行配对。

```
void sendHTML(Connection conn) {
    conn.send("<html>");
    scope(exit) conn.send("</html>");
    ... // 发送头部
    conn.send("<body>");
    scope(exit) conn.send("</body>");
    ... // 发送主体部分
}
```

这个新方案有许多可取的地方。第一，代码布局是线性的，没有额外的嵌套。这种方法可以毫不费力地适应多个“起始/结束”对。第二，对于可能存在的异常抛出所创建的隐藏控制流，利用这种方法便可以不用对 sendHTML 及其可能调用的其他所有函数进行检查。第三，相关的内容都组织在一起，易于理解和维护。第四，这段代码很简洁，因为 scope 语句的符号开销是很低的。

9.7　未捕捉的异常

如果某个异常的处理器未找到，那么会使用一个系统植入的处理器来进行处理，它只会将异常的消息输出到控制台，然后退出，同时还会附上一个非零退出代码。不仅对于传播到 main 之外的异常会发生这种情况，对于 static this 块抛出的异常也一样。

正如前面提到的，正常情况下绝对不要捕捉 Throwable 对象。在某些罕见的情形里，即使是天崩地裂，你也希望可以捕捉到 Throwable，并采取某种应急措施。在整个系统里，你不应指望会存在某种合乎情理的状态。程序的逻辑已被破坏，所以还能进行处理是一件可喜的事情。

自在读书 www.zizidiary.com

第 10 章

契约编程

在我们依赖的各种计算系统里，不管是大型的还是小型的，都离不开我们的大量参与。在这样的大环境下，确保程序的正确性是一个日趋重要的主题。本章介绍的是程序的正确性机制，这种机制是在运行时起作用的（与类型检查和其他语义检查相对，这些检查都是在编译期间实施某些正确性约束）。程序正确性的运行时检查与错误处理只是部分相关联，请不要将它们混淆。更具体地讲，在“事情什么时候会出错”这样一把大伞所覆盖的范围里，它们有着 3 个截然不同的交织区域。

- 错误处理（error handling，第 9 章的主题）解决管理预期运行时错误的技术和惯用法。
- 可靠性工程（reliability engineering）研究的领域是整个系统（如硬件及软件）实现规范的能力（这本书不会讨论可靠性工程）。
- 程序正确性（program correctness）是编程语言研究的领域，它致力于采用静态和动态的方式来证明一个程序按照指定的规范要求是正确的。类型系统是其中最有名的一种用于证明程序正确性的方式（强烈推荐阅读一下 Wadler 的引人入胜的专著 *Proofs are programs* [59]）。本章讨论的是契约编程（contract programming），这是一种增强程序正确性的范型。

程序正确性与错误处理的主要区别在于：后者关心的是包含在程序规范内的错误（例如，如何处理一个损坏的数据文件或无效的用户输入），而前者关心的是编程错误，这些错误将程序的行为置于规范之外（例如，错误计算出一个百分比值，它不在 0～100 这个范围内；或者在某个 `Date` 对象里，意外得到一个负的星期天数）。无视这个重要的差别会导致一些不可宽恕但仍然很普遍的误解，如使用断言（`assert`）来检查文件和网络输入。

契约编程是由 Parnas[45]引入的用来定义软件组件的一种方法，后来随着 Eiffel 编程语言的发展被 Meyer[40]进一步推广。今天的契约编程已经成长为一种流行的软件开发范型。尽管大部分主流编程语言并没有显式地提供对契约编程的支持，但许多机构都拥有实现其基本原则的标准和约定。契约是一个相当活跃的研究领域，近年的工作包括不少高级主题，如用于高阶函数的契约[24]和契约的静态验证[61]。目前，D 语言始终采用的是更为简单和传统的契约编程模式，这也正是本章要讨论的内容。

自在读书
www.zizidiary.com

10.1　契约

契约编程使用了现实生活中的一个隐喻来改善模块化接口的定义和验证。这个隐喻便是具有约束力的契约（binding contract）：当实体 A（如个人、公司）为了 B 的利益而承诺履行某种特定的服务时，A 和 B 之间的合同便会描述成这样：在换取服务过程中，期望 B 提供给 A 的内容；以及 B 一旦履行自己的那部分合同时，A 应承诺提供的内容。

类似地，契约编程范型将函数的规范定义成函数（供应商）与其调用者（客户端）之间的一种合同。此合同的一部分用于规定调用者为调用此函数必须履行的内容要求，而合同的另一部分则规定这个函数在以返回值和/或副作用形式返回时所做的保证。

契约编程的核心概念如下。

- 断言（assertion）：不用绑定到某个特定的函数，断言是一种使用“`if` 可测试条件”的运行时检查。如果这个条件为非零，那么断言不起作用。否则，断言会抛出一个 `AssertError` 对象。`AssertError` 是一个不可恢复异常——它没有继承自 `Exception`，而是直接继承于 `Error`，也就是说，通常情况下不应该捕捉它。
- 前置条件（precondition）：函数前置条件是调用者为了调用该函数必须要履行的所有条件的总和。条件可能与调用点直接相关（如参数值），但也与系统状态相关（如内存的可用性）。
- 后置条件（postcondition）：假设其前置条件都满足的前提下，函数的后置条件为函数在正常返回时做出的所有保证的总和。
- 不变量（invariant）：不变量是一种在某部分计算中始终保持未被修改的条件。在 D 语言里，不变量在方法调用之前和之后总是会引用某个对象的状态。

契约编程很好地概括了某些经过时间考验的概念，今天我们已认为这些概念是理所当然的。例如，函数签名就完完全全是一种契约。在标准库的 `std.math` 模块里，可以找到这样一个函数：

```
double sqrt(double x);
```

整个签名体现了这样一种契约：调用者必须正确提供一个 `double` 类型的值，而函数返回的也是一个 `double` 类型值。你不能调用 `sqrt("hello")`，也不能将 `sqrt` 的结果赋给一个字符串。有意思的是，你可以调用 `sqrt(2)`，虽然 2 是一个 `int` 类型，而不是一个 `double`，这是因为这个签名给了编译器足够的信息，通过将 2 转换成 `double` 类型，便可以帮助调用者满足输入要求。该函数可能会有副作用，如果没有，则可以使用 `pure` 属性来进行指明：

```
// 没有副作用
pure double sqrt(double x);
```

这是一种用于 `sqrt` 的更强且更具有约束力的契约，因为它迫使 `sqrt` 不能有任何副作用。最后，还有 `nothrow` 属性，它允许我们指定一个更为详细的（和更具约束力的）契约：

```
// 没有副作用，绝不会抛出异常
// （在 std.math 里可以找到的真实声明）
pure nothrow double sqrt(double x);
```

自在读书 www.zizidiary.com

现在我们知道，这个函数要么返回一个 double，终止该程序，要么进入一个无限循环。它不会做其他的任何事情。因此，我们在对函数使用契约时，只需要写下其签名即可。

为感受一下函数签名的契约力量，一起来看一小段历史上发生的事情。早期，C 语言的准标准版本（为纪念其发明人 Kernighan 和 Ritchie，它被称为“K&R C”）有一个很怪异的地方。如果完全没对函数进行声明，那么 K&R C 会使用下面的签名来将它当做一个函数：

```
// 如果你没声明 sqrt，但可以调用它，
//     它就像是你声明的一样
int sqrt(...);
```

换句话说，如果你忘了使用# include 来包含头文件 math.h（它提供了 sqrt 的正确签名），也可以调用 sqrt("hello")，而编译器对此是完全不会在乎的。省略号会引入可变参数，这是 C 语言最不安全的特性之一。有一个很奇妙的错误，即有没有包含 math.h 对 sqrt(2) 的调用都能编译通过，但它们却是非常不同的内容。使用#include 时，编译器会在调用 sqrt 之前将 2 转换成 2.0；当没有#include 时，在双方之间会产生一个可怕的误会：调用者发送的是整数 2，而 sqrt 会将它当做一个浮点数，并获得其二进制表示，在 32 位的 IEEE 里，它为 2.8026e-45。ANSI C 意识到此问题的严重性，并通过要求所有函数提供原型来修正这个问题。

函数属性和类型可用来指定简单的契约。属性是以固定方式提供的，而类型在需要的时候很容易进行定义。类型可以在多大程度上描述契约呢？答案有些令人沮丧，即使对于中等复杂程度的契约，（至少在当前技术下）类型还是一个无法胜任的载体。

设计师可以在与函数相关的文档里指定该函数的契约，但我相信大家都会认同这种方案远不能令人满意。组件的用户通常不会去细读它的文档，即使在他们读也很容易犯错误。此外，文档很容易会变得与设计和实现不同步，尤其是在规范很重要而又时常变化的时候（这种情况经常发生）。

契约编程采用了一种更为简单的方法，它以可执行谓词的形式指定契约要求——将契约描述成通过/失败条件的代码片段。现在依次看看这些内容。

10.2 断言

本书已定义过断言（参见 2.3.4.1 节），并且在很多地方都使用到了它——表明这个概念很有用。此外，大部分编程语言都包括某种断言机制，要么是以原语形式，要么是以库形式进行构建。

先回顾一下，通过 assert 表达式可以确保一个表达式在设计上以及在程序无视输入的所有运行过程中都要为非零值：

```
int a, b;
...
assert(a == b);
assert(a == b, "a and b are different");
```

被断言的表达式常常是布尔类型，但也可以拥有任何能进行 if 测试的类型，如数字、数组、类引用或指针。如果表达式为零，那么断言会抛出一个类型为 AssertError 的对象；否

自在读书 www.zizidiary.com

则，什么也不会发生。如果有异常抛出，那么会有一个可选的字符串参数，它是 AssertError 对象所携带的错误消息的一部分。只有在断言失效时，这个字符串才会被计算，这样可以省去一些代价可能很高昂的计算：

```
import std.conv;

void fun() {
   int a, b;
   ...
   assert(a == b);
   assert(a == b, text(a, " and ", b, " are different"));
}
```

函数 std.conv.text 会将其所有参数进行转换并连接成一个 string。这个涉及很多的工作——内存分配、转换以及相关的内容。如果断言成功，还继续做这些工作则是极其浪费的，因此 assert 只会在第一个参数为零时才计算第二个参数。

万一断言失败，那 assert 应该做些什么呢？强行终止应用程序是一种选择（C 语言的同名宏就是这样做的），但 D 语言的 assert 会抛出一个异常。这可不是一个普通的异常，而是一个继承于 Error 的 AssertError 对象——父级异常 Error 在 9.2 节进行过讨论。

assert 抛出的 AssertError 对象会穿过 catch(Exception) 处理器，就像用一把火热的刀子去切开黄油一样。这是一件好事，因为 assert 失败便预示着程序出现了逻辑错误，而在一般情况下是要让逻辑错误尽可能快速有序地终止应用程序。

为捕捉 AssertError 异常，需要在 catch 处理器里直接使用 Error 或 AssertError 来代替 Exception 及其后代。再次提醒：不管怎么样都应尽量避免捕捉 Error。

10.3　前置条件

前置条件是指在进入函数时必须满足的契约责任。例如，假设我们想编写一个契约，它强制要求函数 fun 的输入为非负值。这个便是 fun 施加给调用者的前置条件。在 D 语言里，可以像下面这样编写前置条件：

```
double fun(double x)
in {
   assert(x >= 0);
}
body {
   // 实现 fun
   ...
}
```

in 契约会在函数的 body 之前自动执行。下面还有一个更为简单的版本，不过，当涉及对象与继承时，我们便会看到将前置条件与函数体区分开来是很重要的：

```
double fun(double x) {
   assert(x >= 0);
   // 实现 fun
   ...
}
```

自在读书 www.zizidiary.com

有些语言将契约限制为布尔表达式，并且如果布尔值为假自动抛出异常。例如：

```
// 非 D 语言
double fun(double x)
in (x >= 0)
body {
    ...
}
```

D 语言则更为灵活，因为它允许你检查前置条件，而这些前置条件可没有轻易地将自己限定于单一的布尔表达式。同样地，只要需要，你也可以随意地抛出任何异常，并不仅限于 AssertError 异常。例如，fun 可能会想要抛出某种记录有错误输入的异常类型：

```
import std.conv, std.exception;

class CustomException : Exception {
   private string origin;
   private double val;
   this(string msg, string origin, double val) {
      super(msg);
      this.origin = origin;
      this.val = val;
   }
   override string toString() {
      return text(origin, ": ", super.toString(), val);
   }
}

double fun(double x)
in {
   if (x !>= 0) {
      throw new CustomException("x is less than 0: ", "fun", x);
   }
}
body {
   double y;
   // 实现 fun
   ...
   return y;
}
```

但是请不要滥用这种灵活性。正如前面讨论过的，assert 会抛出 AssertError 对象，它与常规异常不同。当要提醒前置条件出现失败时，最好使用 AssertError 或其他继承于 Error 而非 Exception 的异常。这是因为前置条件的失败表明在程序里有一个严重的逻辑错误，它不应该被随意捕捉。

编译器实际上会采取一些显式步骤来防止契约滥用。第一，在 in 子句里不能执行的 return 语句，这意味着你不能使用契约来完全跳过函数体。第二，D 语言会显式禁止更改契约里的参数。例如，下面的代码是错误的：

```
double fun(double x)
in {
   if (x <= 0) x = 0; // 出错!
      // 在契约里不能修改参数'x'!
}
body {
   double y;
   ...
   return y;
}
```

自在读书 www.zizidiary.com

尽管编译器可以强制要求契约为 pure（这可能是很合乎逻辑的决定），但它不会这样做。这意味着，在契约里，你仍然可以改变全局变量或产生输出。这样做的目的在于：不纯的用法在调试期间比较有用，而如果禁止它们则会让限制过于严格。不过，请一定要记住，在契约里面改变外界的状态在一般情况下并不是一种好的编程风格。契约代码只应该验证契约的遵守情况；如果违反契约，便抛出异常——别管其他的事。

10.4　后置条件

使用 in 契约之后，fun 变得有点不对称，而且在某种程度上可能还是不公平的。fun 只将其要求指定给调用者，但不提供任何保证。调用者为什么要尽量给 fun 提供一个非负数呢？想要检查后置条件，请使用 out 契约。假设 fun 要保证其结果位于 0 和 1 之间：

```
double fun(double x)
// 与之前一样
in {
   assert(x >= 0);
}
// 新增的
out(result) {
   assert(result >= 0 && result <= 1);
}
body {
   // 实现 fun
   double y;
   ...
   return y;
}
```

如果 in 契约或函数的 body 抛出了异常，那么 out 不会被执行。如果 in 契约顺利通过，并且 body 也正常返回，那么 out 契约会被执行。传递给 out 的参数 result 为函数将要返回的任何内容。参数 result 是可选的。out{……}也是一个有效的 out 契约，它不需要结果，或者说是一个返回 void 的函数。在上面的示例里，result 将会是 y 的一个副本。

与 in 契约类似，out 契约只应该进行验证而不会进行修改。out 契约与外部世界的唯一交互应该是：要么什么也不做（通过），要么抛出一个异常（失败）。特别是，请不要将 out 作为在最后进行结果调整的地方。正确的做法是：计算 body 里的结果，并使用 out 来进行检查。下面的代码无法编译通过，其中的原因有两个：out 契约会试着重新绑定 result，也会试着（没有害处但令人有些怀疑）重新绑定到某个实参。

```
int fun(int x)
out(result) {
   x = 42;                          // 出错!
      // 在契约里不能修改参数'x'
   if (result < 0) result = 0; // 出错!
      // 在契约里不能修改结果!
}
body {
   ...
}
```

自在读书 www.zizidiary.com

10.5　不变量

不变量（invariant）指的是在计算期间在某些里程碑处始终能保持满足的一种条件。例如，纯函数能确保程序的整个状态在函数的执行过程中保持完全不会变化。这种保证是很强的，但常常因粒度过粗而很少被使用。

粒度更细的不变保证可被应用于单个的对象，这是 D 语言所使用的模型。例如，有一个简单的 Date 类，它分别以单个整数的形式存储了年、月和日：

```
class Date {
   private uint year, month, day;
   ...
}
```

这样的假设是合理的：在一个Date对象的生命周期里，不会出现year、month和day这3个成员接受无意义值的情况。为表达这种假设，便可以使用invariant：

```
import std.algorithm, std.range;

class Date {
private:
   uint year, month, day;
   invariant() {
      assert(1 <= month && month <= 12);
      switch (day) {
         case 29:
            assert(month != 2 || leapYear(year));
            break;
         case 30:
            assert(month != 2);
            break;
         case 31:
            assert(longMonth(month));
            break;
         default:
            assert(1 <= day && day <= 28);
            break;
      }
      // 对 year 没有限制
   }
   // 辅助函数
   static pure bool leapYear(uint y) {
      return (y % 4) == 0 && (y % 100 || (y % 400) == 0);
   }
   static pure bool longMonth(uint m) {
      return !(m & 1) == (m > 7);
   }
public:
   ...
}
```

针对 30、31 和 29 这 3 个日期的测试能够对 2 月和闰年进行惯例验证。如果某个月有 31 天，那么 longMonth 里的测试会返回 true，而它工作的依据便是“当且仅当一个长月大于 7 月时，它才是一个偶数”，这是有意义的（月份 1、3、5、7、8、10 和 12 都是长月）。

在任何时候对于任何有效的 Date 对象，invariant 都必须要能通过。理论上，只要编译器需要，它都可以发出对 invariant 的调用。然而，事情并非那么简单。例如，假设编译器做出执行决策，要求在每一条语句的末尾插入一个对 invariant 的调用。这不仅会使效率

自在读书 www.zizidiary.com

降低，而且也是错误的。假设要根据另一个 Date 设置某个 Date：

```
// 在类 Date 内部
void copy(Date another) {
  year = another.year;
  __call_invariant();          // 编译器插入内容
  month = another.month;
  __call_invariant();          // 编译器插入内容
  day = another.day;
  __call_invariant();          // 编译器插入内容
}
```

在这些语句之间，Date 很可能会出现短暂不同步的现象，因此每个语句都插入一个 invariant 计算是不正确的。例如，将 2015 年 8 月 1 日这个日期赋值给当前为 2012 年 2 月 29 日的日期会短暂地让这个日期变成 2015 年 2 月 29 日，这是一个无效的日期。

在每一个方法开头和末尾都插入一个 invariant 调用结果会怎么样？假设你编写了一个函数，需要将一个日期提前一个月。这种函数是非常有用的，例如，可用于跟踪每月才发生一次的事件。这个函数只有在月底才需要去关注日的调整，以便让日期跳转，比如 8 月 31 日跳转到 9 月 30 日。

```
// 在类 Date 内部
void nextMonth() {
   __call_invariant();                  // 编译器插入内容
   scope(exit) __call_invariant();      // 编译器插入内容
   if (month == 12) {
      ++year;
      month = 1;
   } else {
      ++month;
      adjustDay();
   }
}
// 辅助函数
private void adjustDay() {
   __call_invariant();                  // 编译器插入内容（很有问题的）

   scope(exit) __call_invariant();      // 编译器插入内容（很有问题的）

   switch (day) {
      case 29:
         if (month == 2 && !leapYear(year)) day = 28;
         break;
      case 30:
         if (month == 2) day = 28 + leapYear(year);
         break;
      case 31:
         if (month == 2) day = 28 + leapYear(year);
         else if (!isLongMonth(month)) day = 30;
         break;
      default:
         // 什么也不做
         break;
   }
}
```

函数 nextMonth 会负责年的翻新，并且会使用一个辅助私有函数 adjustDay 来确保 day 仍然是一个有效的日期。这正是问题所在：在进入 adjustDay 时，不变量可能已被破坏。当然这是可能的——adjustDay 唯一的目的是修正 Date 对象！

是什么让 adjustDay 变得如此特殊呢？是它的保护级别：它是一个 private 函数，因此它只能被其他有权修改 Date 对象的函数访问。一般情况下，当进入和退出 private 函数时，拥有一个已破坏的不变量也是可以接受的。不变量一定会被接受的地方是在 public 方法范围

自在读书 www.zizidiary.com

里：一个对象并不想让客户端的操作发现或放任 this 处于无效状态。

protected 函数怎么样呢？根据 6.7.6 节所讨论的，protected 只比 public 好那么一级。然而，普遍认为在 protected 函数范围里要求满足不变量有些过于苛刻。

如果某个类定义了一个不变量，那么编译器将自动把对不变量的调用插入到下列几个地方：

（1）在所有构造函数的末尾；

（2）在析构函数的开头；

（3）在所有公共非静态方法的开头和末尾。

假设我们戴上了 X 射线透视眼镜，现在便能看被编译器插入到 Date 类里的代码。一起看看这个：

```
class Date {
   private uint day, month, year;
   invariant() { ... }
   this(uint day, uint month, uint year) {
      scope(exit) __call_invariant();
      ...
   }
   ~this() {
      __call_invariant();
      ...
   }
   void somePublicMethod() {
      __call_invariant();
      scope(exit) __call_invariant();
      ...
   }
}
```

有一个关于构造函数和析构函数的细节需要注意。回想一下关于对象生命周期的讨论（参见 6.3 节），即一旦被分配，对象就被认为是有效的。因此，即使某个构造函数抛出了异常，它也一定会将这个对象置于一个持久不变状态。

10.6　跳过契约检查与发行版本

契约在验证应用程序的内部逻辑方面特别重要。可以保证，对于支持契约的编程系统，即使不是全部，也会是绝大部分，它们都支持这样一种模式，即可以忽略它里面的所有契约检查。这种模式应该只有经过彻底审查、验证和测试的程序才能激活。

所有 D 语言编译器都会提供一个标志（在 D 语言的参考编译器里为-release），它可以将契约完全忽略掉。也就是说，对所有契约代码会进行解析和类型检查，但在可执行的二进制文件里不会留下任何痕迹。发布版本在运行时不仅不会进行契约检查（即高风险的），而且也是全速（即运行得更快）。如果应用程序将所有因素都考虑进去，那么在跳过契约检查之后所增加的风险是很低的，而且这种风险能换得速度的提高，这是很值得做的事情。程序运行中可能不带契约，这敲响了一个警钟：代码不应该将契约用于那些可能是“合理失败”的例行检查。契约必须要给无法预料的错误预留空间，而这些错误所反映的是程序里的逻辑错漏。同样，也不应把契约用来确保用户输入的正确性。另外，是否还记得这样一些被再三提醒的警告呢？即不要在 assert、in 和 out 里做任何有意义的事情（如副作用）。到这里，原因是显而易见的：如果某个程序采取这种

自在读书 www.zizidiary.com

令人讨厌的动作，那么它在的非发行与发行这两种完全不同的模式里会表现出非常奇怪的行为。

有一个常见的错误是带有副作用的断言表达式，如 assert(++x < y)，这个势必会让人抓破头皮。这样的情况是最糟的：错漏在发行模式里自然便会显现出来，很明显这时没有多少方法可供你任意选择，用来找出问题的根源。

10.6.1　enforce 不（完全）是 assert

很遗憾 assert 不会出现在发布版本里，即使它用起来很方便。

```
if (!expr1) throw new SomeException;
...
if (!expr2) throw new SomeException;
...
if (!expr3) throw new SomeException;
```

上面的写法可以替换成下面的样子：

```
assert(expr1);
...
assert(expr2);
...
assert(expr3);
```

鉴于 assert 如此简洁，许多库都提供一个“总是断言”的特性。它会检查一个条件，并且如果这个条件为 0，那么不管是否是以发行模式编译，它都会抛出异常。这样的检查器在 C++ 里被命名为 VERIFY、ASSERT_ALWAYS 或者 ENFORCE。D 语言在 std.exception 模块里也定义了一个这样的函数，其名字为 enforce。使用 enforce 的语法与 assert 相同：

```
enforce(expr1);
enforce(expr2, "That isn't quite true");
```

如果传入的表达式为 0，那么 enforce 会抛出一个类型为 Exception 的对象，不管你是以在发布模式还是以非发布模式编译程序，结果都一样。如果想要抛出一个不同类型的异常，那么可以像下面这样来指定它：

```
import std.exception;
bool something = true;

enforce(something, new Error("Something isn't right"));
```

如果 something 为 0，那么第二个参数会被抛出；而 enforce 会以惰性方式计算这个参数，这样当 expr1 为非零值时，不会发生创建对象的操作。

尽管 assert 和 enforce 的外观很像，但它们从根本上讲是服务于不同的目的的。此外它们还有以下几点不同之处。

- assert 检查应用程序的逻辑，而 enforce 检查那些不会威胁到应用程序完整性的错误条件。
- assert 只会抛出不可恢复的 AssertError 异常，而 enforce 默认情况会抛出一个可恢复的异常（并且可以抛出任何带有一个额外参数的异常）。
- assert 可能会消失，在考虑函数流的时候，请不要将其包括在内；enforce 绝不会消失，在调用 enforce(e) 之后，可以假定 e 是非零值。

自在读书 www.zizidiary.com

10.6.2 assert(false)

针对在编译期间已知为 0 的常量的断言，如 assert(false)、assert(0)或者 assert(null)，其行为与普通断言有点儿不一样。

在非发行模式下，assert(false)不会采取任何特殊的动作，它只会抛出一个 AssertError 异常。

不过，在发行模式下，assert(false)在编译后依然存在，它总是会导致程序停止。只是这次不会出现异常，并且在遇到 assert(false)之后，也没机会继续运行下去。这个程序最终会崩溃。在 Intel 机器上，它是通过执行 HLT（即 halt，停止）指令实现的，这条指令能让程序立即终止。

我们中有许多人倾向于将崩溃认作是一件极度危险的事件，并且认为这种事件表明程序已失去控制。这种倾向之所以普遍，很可能是因为许多失去控制的程序迟早会以崩溃的方式终止。但 assert(false)是以一种十分可控的方式来终止程序。事实上，在某些操作系统里，HLT 会自动加载调试器，并将它定位到正好是触发崩溃的那个 assert 所在的位置。

assert(false)的这种特殊行为，其目的何在？一种比较明显的用法与系统级程序有关。必须要有一种可移植的方式来触发 HLT，并且 assert(false)需要与编程语言的其他部分紧密结合。此外，编译器还要了解 assert(false)的语义，例如，它不允许在表达式 assert(false)之后出现死代码：

```
int fun(int x) {
    ++x;
    assert(false);
    return x; // 错误!
              // 语句是不可到达的!
}
```

另一方面，在其他一些情况里，你可以通过添加assert(false)禁止显示编译器错误。例如，前面讨论过的标准库函数std.exception.enforce(false)，可以像下面这样调用它：

```
import std.exception;

string fun() {
    ...
    enforce(false, "can't continue"); // 总是抛出异常
    assert(false);                    // 不可到达
}
```

调用 enforce(false)总是会抛出一个异常，但是编译器并不知道它。为了让编译器明白那个点是不可到达的，必须要插入一个 assert(false)。fun 在结束时执行语句 return 也是可以的，但在这种情形下，如果有人注释掉 enforce 调用，fun 便会开始返回一些虚假值。assert(false)是个名副其实的“好帮手”，它能将你的代码从这种状况中拯救出来。

10.7 契约：请不要用于洗涤输入

本节将讨论一件存在争议且与契约有关的事情，它是争论持续不断的源头。此事本质上可

自在读书 www.zizidiary.com

归结为这样一个问题：如果一个函数一定要做某种检查，那么这个检查应该放置在哪里——是在契约里，还是在函数体里？

当第一次对契约编程有所习惯时，我们中的许多人会试图将大部分检查都挪到契约里去。例如，有一个叫 readText 函数，它会以字符串的形式完整地加载一个文本文件。利用契约，我们可以定义出下面的内容：

```
import std.file, std.utf;

string readText(in char[] filename)
out(result) {
   std.utf.validate(result);
}
body {
   return cast(string) read(filename);
}
```

readText 实际上是标准库里的一个函数，它可以在模块 std.file 里查找到。

readText 依赖于其他两个文件函数。首先，它使用 read 来将整个文件加载到内存缓冲区里。内存缓冲区拥有的类型为 void[]，readText 会将它强制转换成字符串类型。但是这种做法是不正确的：如果文件包含了畸形的 UTF 字符，结果会怎么样呢？为验证那个类型转换操作，out 契约可以通过调用 std.utf.validate 来验证这个结果。如果缓冲区包含了无效的 UTF 字符，那么它会抛出一个 UtfException 对象。

这样也不错，只是这里还有一个很重要的问题：契约必须要验证应用程序的逻辑，而不是验证其输入的有效性。任何不考虑应用程序特有问题的内容都不属于契约的范围。同样地，不要期望契约去改变应用程序的语义——因此，D 语言故意对可以在契约中进行修改的内容进行了限制。

假设契约都不会失败，那么在有和没有契约的情况下，应用程序运行的行为和结果一定是相同的。这是一个非常简单和容易记住的“试金石”，它可用于判定一个内容是否为契约。契约是规范性检查，在正确实现时，取消这种检查也不会让这个实现无法工作！这就是契约如何发挥作用的地方。“期望文件总是有效的”，它可能显露出了某种积极的态度，但不应成为 readText 规范的一部分。正确的 readText 定义是让这种检查成为函数的组成部分：

```
import std.file, std.utf;

string readText(in char[] filename) {
   auto result = cast(string) read(filename);
   std.utf.validate(result);
   return result;
}
```

根据前面的讨论，关于检查位置这个问题的答案就是：如果这种检查关心的是应用程序的逻辑，那么它应该放置在契约里；否则，这种检查应该放置在函数体里，并且不能被跳过。

这个听起来很不错，但在单独构建的应用程序里，即在使用独立实体编写的通用库里，“应用程序逻辑”该如何定义呢？假设有一个大型通用库，如微软的 Windows API 或者 K 桌面环境（KDE）。许多应用程序都会使用这样的 API，不可避免地会出现这样的情况，即库函数接收到的参数是不符合规范的（事实上，操作系统的 API 并不怕收到各种畸形的参数）。如果某个应用程序并不满足库函数调用的前置条件，那么该去指责谁呢？这显然是应用程序的错，但是掌握主动

自在读书 www.zizidiary.com

权的是库——其中涉及的内容包括不稳定性（instability）、未定义行为、库内部的状态损坏、崩溃，所有那些不好的事情。这个看起来很不公平，与在错漏百出的使用这个库的应用程序里的表现相比，这些问题在库里的表现更为糟糕（“库 Xyz 很不稳定，常会发生让人意外的怪事”）。

一套通用的和分布广泛的 API，应该将对所有函数的所有输入的验证当成是一件理所当然的事——而不是将其置于契约里。参数验证失败毫无疑问是库的错漏。没有哪位发言人会一边不停地翻动书本或文献，一边说：“我们全面使用了契约编程，因此我们没有错误。”

函数应该使用前置条件来指定像参数范围这样一些内容，这种观点是否变得无效了呢？并非如此。这是将“应用程序逻辑”与“用户输入”进行定义和区分的事。对属于应用程序组成部分的函数来说，接收有效的参数是应用程序逻辑的一部分；而对属于独立分发库的通用函数而言，参数仅仅是用户输入而已。

另一方面，让库在其私有函数里使用契约也是不错的做法。这些函数都涉及库的内部工作方式，并且无法被用户代码访问到，因此让它们使用契约来表达对规范的遵守是很明智的。

10.8　契约与继承

常被引用的 Liskov 替换原则（LSP）[38]指出：继承就是可替代性，即在所有期望是基类对象的地方，一定能使用派生类对象进行替换。这种见识在本质上决定了契约与继承的相互影响。

在现实世界里，契约与置换之间的联系为：一旦建立了契约，替代承包商最低限度必须要完成这项工作，在指定的最小容忍范围内交付工作，并且最多只能要求在契约里确立的同额报酬。当然也存在一定的变数，但绝不会朝着收紧契约的前置条件或放宽后置条件这样的方向变化。如果出现了这样的情况，那么这个契约便是无效的，必须要重写。这种变数只会关心契约里的那些不会对理解产生负面影响的变化：替换是允许的，但只能提更少的要求，同时承担更多的责任。

10.8.1　继承与 in 契约

再来看看 Date 这个示例。现在假设要定义一个非常简单的、轻量级的 BasicDate 类。它仅提供最小的支持，并且把增强部分留给派生类去完成。BasicDate 要提供函数 format，它会接受一个用来表示格式规范的字符串，并且会返回一个被正确格式化的日期字符串：

```
import std.conv;

class BasicDate {
   private uint day, month, year;
   string format(string spec)
   in {
      // 要求 spec 等于"%Y/%m/%d"
      assert(spec == "%Y/%m/%d");
   }
   body {
      // 简单的实现
      return text(year, '/', month, '/', day);
   }
   ...
}
```

自在读书 www.zizidiary.com

Date.format 实施的契约要求格式规范为%Y/%m/%d，它表示的含义是“长格式的年，后面紧跟着一根斜线、月份、一根斜线、天”。这是 BasicDate 唯一要求被支持的格式。派生类可以添加本地化、国际化，以及相关的内容。

继承于 BasicDate 的类 Date 想要提供一个更好的 format 原语。例如，Date 想让说明符%Y、%m 和%d 可以出现在任何位置，并且能与任意字符进行混合。同时，也允许使用%%，因为它代表的实际字符是%。也允许相同的说明符重复出现。为实施所有这些要求，Date 编写了自己的契约：

```
import std.regex;

class Date : BasicDate {
   override string format(string spec)
   in {
      auto pattern = regex("(%[mdY%]|[^%])*");
      assert(!match(spec, pattern).empty);
   }
   body {
      string result;
      ...
      return result;
   }
   ...
}
```

Date 借助于一个正则表达式，将其约束施加到了 spec 上面。正则表达式在字符串操作中有着不可估量的作用。强烈推荐大家阅读 Friedl 的经典著作 *Mastering Regular Expressions*[26]。这里不会对正则表达式进行深入讨论，但对于(%[mdY%]|[^%])*，其实际的含义是“在一个%之后紧跟有 m、d、Y 或%中的任何一个；或者除%以外的任何内容——它可以重复零到多次”。通过手工实现的用于匹配此模式的等价代码将会相当冗长。这里的 assert 会确保：在模式 pattern 与字符串 spec 进行匹配时，返回的不是一个空匹配，即它是起作用的。更多关于在 D 语言里使用正则表达式的内容，请阅读标准模块 std.regex 的在线文档。

Date.format 的总契约是什么？它应该要考虑 BasicDate.format 的契约呢，还是不用管它？如果父级的 in 契约失败的话，也没什么问题，只要派生类的 in 契约通过就行。同时，Date.format 的契约绝不应该加强 BasicDate.format 的 in 契约。新冒出的规则如下：在某个被改写的方法里，首先会执行基类的契约。如果成功，则将控制转移至函数体。否则，执行派生类的契约。如果成功，则将控制转移至函数体。否则，报告契约失效。

换句话说，in 契约会采用具有短路效果的析取方式进行组合：恰好会有一个一定能通过，并且首先会尝试基类契约。这样，才不会出现派生类的契约比基类的契约更难以满足的情况。与之相反的是，派生类还为失败的前置条件提供了第二次机会。

对于 Date 和 BasicDate，上面的规则非常有用。首先，组合契约会对精确模式"%Y/%m/%d"进行检查。如果成功，则进行格式化。如果失败，那么就按照派生结构，先检查更宽松的契约。如果通过，则格式化可以继续进行下去。

契约组合在一起所生成的代码有点像下面这个样子：

```
void __in_contract_Date_format(string spec) {
   try {
      // 尝试基类的契约
      this.BasicDate.__in_contract_format(spec);
   } catch (Throwable) {
```

自在读书 www.zizidiary.com

```
        // 基类契约失败，尝试派生类的契约
        this.Date.__in_contract_format(spec);
    }
    // 成功，可以调用函数体
}
```

10.8.2　继承与 out 契约

对于 out 契约，情况恰恰相反：当使用派生对象替换基对象时，被覆盖的函数必须提供比契约承诺部分更多的内容。因此，那个改写方法必须要始终实现基类的 out 保证（与 in 契约的情况不一样）。

于是乎，这意味着基类应该将契约设置得尽可能宽松，以避免出现派生类跳过约束的风险。例如，如果 BasicDate.format 强制要求返回字符串格式为“年/月/日”，那么所有派生类都将无法实现任何其他的格式化操作。也许 BasicDate.format 可以要求一个更弱一些的契约，例如，如果格式化字符串不为空，那么不允许输出一个空的字符串：

```
import std.range, std.string;

class BasicDate {
   private uint day, month, year;
   string format(string spec)
   out(result) {
      assert(!result.empty || spec.empty);
   }
   body {
      return std.string.format("%04s/%02s/%02s", year, month, day);
   }
   ...
}
```

Date 可以将其目标调高一些：它可以根据格式规范计算出预计的结果长度，然后将实际结果的长度与那个预计长度进行比较。

```
import std.algorithm, std.regex;

class Date : BasicDate {
   override string format(string spec)
   out(result) {
      bool escaping;
      size_t expectedLength;
      foreach (c; spec) {
         switch (c) {
            case '%':
               if (escaping) {
                  ++expectedLength;
                  escaping = false;
               } else {
                  escaping = true;
               }
               break;
            case 'Y':
               if (escaping) {
                  expectedLength += 4;
                  escaping = false;
               }
               break;
            case 'm': case 'd':
               if (escaping) {
                  expectedLength += 2;
```

自在读书 www.zizidiary.com

```
                escaping = false;
            }
            break;
        default:
            assert(!escaping);
            ++expectedLength;
            break;
        }
    }
    assert(walkLength(result) == expectedLength);
}
body {
    string result;
    ...
    return result;
}
...
}
```

为什么要使用walkLength(result)，而不用result.length呢？在UTF编码的字符串里，字符的数量可能会比它的 char 长度小。对于这两个 out 契约，正确组合后的 out 契约是什么呢？答案很简单：基类的契约也必须要验证。因此，如果派生类承诺有额外的契约义务，则这些契约也必须要满足。这是一个简单的结合。下面是编译器在将基类和派生类的契约组合后可能生成的代码：

```
void __out_contract_Date_format(string spec) {
    this.BasicDate.__out_contract_format(spec);
    this.Date.__out_contract_format(spec);
    // 成功
}
```

10.8.3　继承与 invariant 契约

跟 out 契约的情况一样，我们一直在寻找某种结合方式，即一种“与”关系：一个类除了满足自己的不变量以外，还必须要满足基类的所有不变量。对于一个类来说，它没办法削弱基类的不变量。当前的编译器会从继承层次的最高层往下调用所有的 invariant() 子句，但对于 invariant 的实现者没什么要求。如前所述，不变量不应该有副作用。

10.9　接口里的契约

契约最有意思的应用可能是与接口相结合。接口就是一个复杂的契约，因此每一个接口方法都应该描述一个抽象契约（没有 body 的契约），这种说法非常贴切。这种契约会按照接口定义的但仍未实现的原语被强制执行。

例如，我们对 6.14 节定义的 Stack 接口进行增强。参考实现如下：

```
interface Stack(T) {
    @property bool empty();
    @property ref T top();
    void push(T value);
    void pop();
}
```

现在往接口里附带一些契约，这个接口揭示了发这些原语之间的相互作用。接口契约看起

自在读书 www.zizidiary.com

来跟普通的没有 body 的契约是一样的。

```
interface Stack(T) {
    @property bool empty();

    @property ref T top()
    in {
        assert(!empty);
    }

    void push(T value)
    out {
        assert(value == top);
    }

    void pop()
    in {
        assert(!empty);
    }
}
```

对于一个带有契约的接口方法，末尾的分号不再是必要的。利用这个新定义的接口 Stack，所有基于它的实现都只会在 Stack 的契约定义的范围内起作用。这样会有一个好处，即使用契约增强的 Stack 是一种很好的栈规范，同时易于被程序员阅读和进行动态验证。

正如在 10.7 节所讨论的，Stack 的契约可能会被编译器排除在外。如果你定义了一个适合大型应用和具有通用性的容器库，那么将方法调用当做用户输入可能是个不错的主意。在这种情况下，则可能更适合使用 NVI 惯用法（参见 6.9.1 节）。使用 NVI 对有效调用频繁进行检查的栈接口像下面这个样子：

```
interface NVIStack(T) {
protected:
    ref T topImpl();
    void pushImpl(T value);
    void popImpl();

public:
    @property bool empty();

    final @property ref T top() {
        enforce(!empty);
        return topImpl();
    }

    final void push(T value) {
        pushImpl(value);
        enforce(value == topImpl());
    }

    final void pop() {
        enforce(!empty);
        popImpl();
    }
}
```

NVIStack 大量使用了 enforce（一种编译后仍然存在的测试），而且把 push、pop 和 top 定义为 final 类型，因此它不可能通过这些实现劫持。这便达到了一种不错的效果，即所有主要的错误处理已依次从各个具体实现提升到了接口——这是一种很好的重用形式和责任划分形式。NVIStack 的实现可以放心地假定：在有效的状态里，pushImpl、popImpl 和 topImpl 总会得到调用，并对它们相应地进行优化。

自在读书 www.zizidiary.com

第11章

扩展性

有一条格言是这样说的：如果一个有100行的程序，即使它会破坏所有的优秀编程规则，也可以工作起来。这条格言表达出的只是一个缩影——在实际编写一个有10 000行的程序时，可能会在小范围内对细节加以注意，但对正确进行大规模、模块化开发的规则却未放在心上。甚至有些上百万行的项目也会破坏某些适合于大规模设计的规则。

在软件工程中，许多牢固的原则也是它们的缩影的体现。在一个小模块里或者当将整个应用程序连接在一起时，关注点的分离和信息隐藏都同样会起到作用。然而，这些原则的具体体现会随着这些原则应用规模的不同而有所差别。本章关注的内容是更大实体（如整个文件、目录、库和程序）的组合。

针对大规模的模块化，D语言在定义其方法时遵循了几条检验可靠的原则，并且在符号查找的方式上，它也带来了几点很有意义的创新。

11.1 包和模块

编译、保护和封装的单位是物理文件。多个文件打包的单位是目录。而这一点正变得日趋成熟。站在模块化的角度来看，我们将D语言源文件称为模块（module），而将包含D语言源文件的目录称作包（package）。

老实说，在一个出色的数据库里，程序源代码真的会感觉好一些。D语言使用的是一个我们长时间锤炼过的“数据库”，它能够很好地与多种内容集成在一起，其中包括安全特性、版本控制、备份、操作系统级的保护、日志记录以及所有你能想的；而且D语言也为进行大规模开发构筑了一个很低的准入门槛，因为开发需要的基本工具是编辑器和编译器。

一个D语言模块就是一个带有“.d”或“.di”后缀的文本文件。D语言工具链不会基于它们的后缀名区别对待，但一般的习惯是：实现代码放置在“.d”文件里，而接口代码放置在“.di”（即D语言接口的意思）文件里。源文件必须以UTF-8、UTF-16或UTF-32进行编码。

自在读书
www.zizidiary.com

源文件里的字节顺序（UTF-16 或 UTF-32 才会有的情况）可以通过文件最开始的几个字节进行检测，这是一个很小的标准协议，即 BOM（byte order mark，字节序标志）。表 11-1 显示了 D 语言编译器是如何标识源文件编码的，它遵循 Unicode 标准[56，第 2 章]。

表 11-1　用于区分 D 语言源文件的字节序标志（BOM）。这些模式尝试的顺序为自上而下，首次测试成功那个即为文件的编码。xx 为任何非零字节值

开始字节	文件编码	这些字节可抛弃吗
00 00 FE FF	UTF-32 高位优先	✔
FF FE 00 00	UTF-32 低位优先	✔
FE FF	UTF-16 高位优先	✔
FF FE	UTF-16 低位优先	✔
00 00 00 xx	UTF-32 高位优先	
xx 00 00 00	UTF-32 低位优先	
00 xx	UTF-16 高位优先	
xx 00	UTF-16 低位优先	
其他内容	UTF-8	

有些文件缺少 BOM，但 D 语言有一种很明确的手段，可以用来自动检测编码。这个自动检测程序巧妙地利用了这样一个事实：任何格式良好的D语言模块一定以某些ASCII字符作为开头，即 Unicode 码点都小于 128。这是因为，按照 D 语言语法，格式良好的模块，其开始部分一定是 D 语言关键字（基于 ASCII）、ASCII 空白符、注释起始符“/”或者是一些以“#”开头的指令，它们一定都是 ASCII 字符。如果按自上而下的顺序对表 11-1 里的模式进行尝试，那么根据首次匹配到那个模式即可明确地推断出具体的编码。如果推断出的编码是错误的，那么也不会产生什么害处——很明显，这个文件就是错误的，因为它的起始字符不是有效的 D 语言代码。

如果某个源文件的前两个字符（在跳过字节序标志之后，如果这种标志存在的话）是“#!”，那么这些字符加上第一个换行符“\n”之前的所有字符会被忽略。这样做的目的在于允许使用系统所支持的“shebang”（井号加叹号）特性。

11.1.1　import 声明

在前几章的代码里，为访问标准库里的内容，它们常常带有 import 语句，例如：

```
import std.stdio; // 访问 writeln 等内容。
```

为导入某个模块，需要在import声明里指定这个模块的名称。模块名包含的必须是相对路径，而相对的起始点就是进行编译的那个目录。例如，有一个目录层次结构，如图11-1所示。

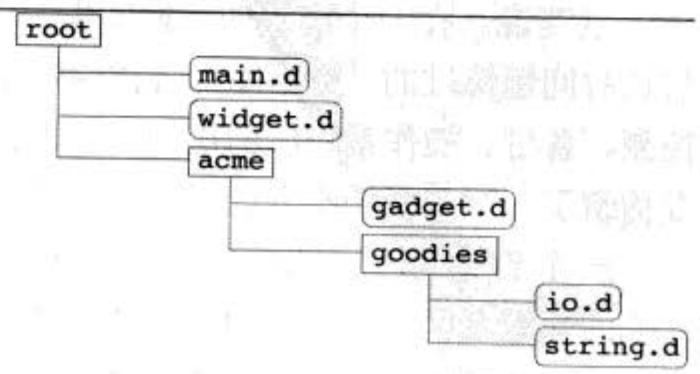


图 11-1　一个样本目录结构

我们假定编译发生在 root 目录里。为访问文件 widget.d 里的定义，所有其他文件都必须要在声明的顶层包含下面的内容：

自在读书 www.zizidiary.com

```
import widget;
```

“顶层”(top level)表示的是所有作用域(如函数、类和结构)之外。当编译器遇到 import 声明时,它会从 root 目录开始查找 widget.di(首先)或 widget.d(其次),在找到 widget.d 之后便导入它的符号。为使用目录层次里更深一些的文件，项目里的所有其他文件都必须导入起始于 root 的相对路径，同时使用“.”作为分隔符:

```
import acme.gadget;
import acme.goodies.io;
```

在 import 声明里,我们也常使用逗号分隔的列表。上面的两个声明与下面的声明是等价的:

```
import acme.gadget, acme.goodies.io;
```

请注意，一个深处目录层次内部的文件，如 gadget.d，在导入其他文件时仍然是相对于编译开始的地方，即 root，而不是相对于 gadget.d 自己的位置。例如，为访问到 io.d 里符号，gadget.d 必须要使用

```
import acme.goodies.io;
```

而不能使用

```
import goodies.io;
```

再看一个示例,如果 io.d 使用了 string.d,那么它需要导入 acme.goodies. string，即使这两个文件在同一个目录里也一样。当然，所有这些都是假设编译开始于 root 目录。如果你决定转到 acme 目录，然后在那里编译 gadget.d，那么它必须要导入 goodies.io。

模块导入的顺序无关紧要。D 语言设想的是这样一种方式，即模块的语义与导入其他模块的顺序是无关的。

import 声明只接受符号，因此在 D 语言里包和模块的名称都必须是有效的 D 语言符号(参见 2.1 节)。例如，有一个名为 5th_element.d 的文件，那么你便无法将其导入到另一个模块里，因为 5th_element 并不是一个有效的 D 语言符号。同样地，如果将文件放置到一个名为 input-output 的目录里，那么你也不能将这个目录当做一个在 D 语言包来使用。总之，所有的 D 语言源文件和目录，其名字都必须是有效符号。另有一个额外的习惯，即所有的包和模块名都最好是小写的，在对于文件名的大小写并不严格区分的操作系统里，这样做可以避免产生混淆。

11.1.2　模块搜索根目录

当编译器解析一个 import 声明时，它并不仅仅从编译发生的当前目录开始搜索。如果真是这样，那就无法使用任何标准库和任何其他部署在当前项目目录之外的库了。要知道，我们总是会从 std 包里导入模块，并且我们的项目并没有像 std 这样的子目录。这种机制是如何工作的呢?

像许多其他语言一样，D 语言允许你设置一些根目录，在那里可以进行模块搜索操作。有一个可传递给编译器的命令行参数，它允许添加任意数量的目录，并将它们当做模块搜索的根目录。准确实现此操作的语法与具体的编译器有关。参考编译器 dmd 使用的命令行开关为-I，其后再紧跟一个路径。例如，在 Windows 平台里可以使用的-Ic:\Programs\dmd\src\phobos，或者在 Unix 平台里使

自在读书 www.zizidiary.com

用的-I/usr/local/src/phobos。使用附加的-I 开关便可以往根目录列表里添加任意数量的路径。

例如，为解析 import path.to.file，编译器首先会在当前目录里搜索子目录 path/to[①]。如果这个目录存在，再查询文件 file.di。如果它未找到，那么查询文件 file.d。如果该文件被找到，那么搜索结束。否则，对使用-I 传入的每一个目录执行类似的搜索。在模块第一次被找到时，搜索会立即停止。如果遍历完所有目录也没有找到此模块，那么会出现一个“模块没有找到”的错误，然后中止编译。

如果组合 path.to.不存在，那么编译器会直接在所有根目录里搜索这个模块。

仅为了访问标准库或其他广泛使用的库，而要求用户添加一个命令行开关，这将是一件很麻烦的事情。正因为如此，参考编译器 dmd（实际上对于其他编译器也一样）会使用一个简单的配置文件，让它包含一些默认的会自动附加到任何命令行的开关。安装过程必须要对配置文件进行适当的设置，以便能找到运行时支持库和标准库。因此，如果只是输入下面的内容：

```
% dmd main.d
```

那么在命令行不需要添加任何内容便可以找到标准库的内容。为详细地查看到每一个模块到底是在哪里被找到的，可以在 dmd 的命令行里添加一个-v（详细输出）标志。更多关于 D 语言安装版本是如何加载配置选项的信息，可以查阅编译器的实现文档（关于 dmd 的在线文档，请参考[18，19，20，21]）。

11.1.3 名字查找

看起来可能有点儿奇怪，D 语言没有全局作用域和全局命名空间。特别是，没方法定义一个真正的全局对象、函数以及类名。这是因为定义这种实体的唯一方式就是将它放在一个模块里，而任何模块都有名字。于是，这个模块的名字便会引入一个命名作用域。即使所有类对象的根——Object，也不是一个真正的全局名字，因为它实际上是被编译器安装版本提供的模块 object 以 object.Object 的形式引入的。例如，假设 widget.d 有如下内容：

```
// 这里是 widget.d
void fun(int x) {
   ...
}
```

fun 的定义不会引入全局可用的符号 fun。相反，无论谁（如 main.d）导入 widget 模块，它都可以获得对符号 widget.fun 的访问：

```
// 这里是 main.d
import widget;

void main() {
   widget.fun(10); // 正确。在 widget 模块里查找函数
}
```

这个非常好，也很模块化，但也相当冗长和过于严格。当请求 fun 时，如果其他模块并没有定义函数 fun，难道编译器不能决定 widget.fun 作为唯一的竞争者刚好就成为那个获胜者吗？

① 本段文本使用“/”作为通用的路径分隔符，要知道实际的分离符是取决于操作系统的。

自在读书 www.zizidiary.com

事实上，名字查找正是这样工作的。每一个被导入的模块都会提供一个独立的命名空间，但当有符号被找到时，便执行下面的步骤。

（1）在当前的作用域里查找符号。如果找到，则查找成功结束。

（2）在当前模块的作用域里查找符号。如果找到，则查找成功结束。

（3）在所有导入的模块里查找符号。

- 如果未找到，则失败。
- 如果恰好在某个模块里找到，则查找成功结束。
- 如果在不止一个模块里找到，并且如果这个符号不是一个函数名，那么会给出“重复的符号”这样的错误信息来提示失败。
- 如果在不止一个模块里找到，并且如果这个符号是一个函数名，那么应用跨模块的重载解析（参见 5.5.2 节）。

这种方法的效果很好，即它允许客户端代码在大部分的时间里是简洁的，而只在需要时才变得稍微有些冗长。在前面的示例里，main.d 可以在不带任何装饰的情况下简单地调用 fun：

```
// 这里是 main.d
import widget;

void main() {
    fun(10); // 正确，只有 widget 模块定义了 fun
}
```

假设有一个文件 io.d 也定义了一个名叫 fun 的函数，其类似的签名如下：

```
// 这里是 io.d，这位于目录 acme/goodies
void fun(long n) {
    ...
}
```

接着，在 main 里导入 widget.d 和 io.d。然后，在不进行装饰的情况下使用 fun 便会出现错误，但通过模块名进行限定后的调用就可以正常工作：

```
// 这里是 main.d
import widget, acme.goodies.io;

void main() {
    fun(10); // 错误!
             // 有歧义的函数 fun()
             // 它可以在 widget 和 acme.goodies.io 里找到
    widget.fun(10);           // 正确，显式区分
    acme.goodies.io.fun(10); // 正确，显式区分
}
```

请注意，这种歧义是潜在的。如果从未以模棱两可的方式去查找某个符号，那么编译器也绝不会抱怨。

跨模块的函数重载

第 5 章对函数重载如何实现跨模块应用进行了讨论（参见 5.5.2 节），并给出了一个示例。在该示例中的某些模块定义了一个同名函数，但没有产生任何歧义。既然我们已对模块和模块化的内容有了更多的了解，那么现在完全可以结束那个讨论了。

函数劫持（function hijacking）与模块化原则有点背道而驰。如果模块里的函数出现竞争，

自在读书 www.zizidiary.com

并且源自另一个不同模块的函数会接管调用，那么这时便会发生函数劫持。典型的表现是一个工作模块会完成不同的事情，而具体结果却取决于其他模块的包含内容或者模块的包含顺序。

函数劫持的表现就是不一样的结果与预设规则错误地交织在一起的时候。例如，在上面示例里，widget 定义了 fun(int)，而 acme.goodies.io 定义的是 fun(long)。于是，在 main 里的调用 fun(10) 应该转移到 widget.fun，这似乎是合情合理的，因为这个匹配“更适合”。然而，这只是其中的一种情形，即“更适合”与“适合”存在敌对关系。如果 main 只导入 acme.goodies.io，那么 fun(10) 自然会转移到唯一候选项 acme.goodies.io.fun。然而，如果这时 widget 出现了，那么 fun(10) 会突然转移到 widget.fun。事实上，widget 会按照某个契约进行仲裁和分担，而这个契约最初是在 main 和 acme.goodies.io 之间达成的——这是一种很可怕的与模块化原则相违背的行为。

难怪编程语言都要回避函数劫持。C++支持函数劫持，但大部分样式指南都建议避免使用它；Python 和许多其他语言根本就不允许任何劫持。另一方面，过多的逃避可能会导致规则过于严格，从而造成名字里的符号串变得很长。

D 语言以一种原始的方式来解决函数劫持问题。指导 D 语言实现跨模块重载的基本原则就是添加或删除 import 不会改变函数名解析的决策。使用 import 可能会导致可编译模块无法编译，而不可编译的模块变得可以编译。所消除的危险情况是：使用 import 时，程序编译是没有问题的，但却会出现不同的“重载解析”（overloading resolution）。

在模块里找到的任何函数调用，如果函数名出现在多个模块里，并且这个调用在这两个模块里都可以工作，那么此调用就是错误的。另一方面，如果只有一个解析能形成有效的函数调用，那么此调用是合法的，因为它不会出现函数劫持的风险。

在这个示例里，如果 widget 定义了 fun(int)，而 acme.goodies.io 定义的是 fun(long)，那么在模块 main 里的状态将会是下面这个样子：

```
import widget, acme.goodies.io;

void main() {
    fun(10);        // 错误！有歧义的跨模块重载！
    fun(10L);       // 正确，毫无歧义地转到 acme.goodies.io.fun
    fun("10");      // 错误！无匹配！
}
```

在 import 行里，添加或删除 widget 或 acme.goodies.io 中的某一个，都可能会使一个已遭破坏的程序仍然可以工作，或者破坏一个可以工作的程序，或者让可以工作的程序仍然可以工作——在最后一种情况里，对 fun 的调用从不需要做任何抉择。

11.1.4　public import 声明

默认情况下，在导入模块里的符号查找不具有传递性。也就是说，在图 11-1 所示的那个目录层次结构里，如果模块 main 导入模块 widget，而模块 widget 依次又导入模块 acme.gadget，那么从 main 开始查找符号时并不会搜索模块 acme.gadget。无论模块 widget 导入什么内容，都只是 widget 的实现细节，main 对此并不关心。

自在读书 www.zizidiary.com

但是，模块 widget 有时是另一个模块的增强，或者只有与另一个模块进行结合才具有意义。例如，widget 的定义可能会使用和需要很多 acme.goodies.io 里的定义，对于任何其他模块来说，在没有导入 acme.goodies.io 时使用 widget 是无用的。在这种情况下，使用 public import 声明可以为客户端代码提供帮助：

```
// 这里是 widget.d
// 让 acme.goodies.io 里的符号对于 widget 的客户端代码都是可见的
public import acme.goodies.io;
```

上面的 public import 声明会让导入 widget.d 的模块看到所有定义在 acme/goodies/io.d 的符号，它们仿佛就是 widget.d 自己定义的一样。事实上，在 widget.d 里，public import 会为定义在 io.d 中的每一个符号添加一个 alias 声明（源代码或目标代码都不会重复，只有各种符号的重复）。假设 io.d 定义了函数 print(string)，并且在 main.d 输入下面的代码：

```
import widget;

void main() {
   print("Hello");                  // 正确，查找到 print
   widget.print("Hello");           // 正确，widget 实际上定义了 print
}
```

假若又从 main 里导入 acme.goodies.io，结果会怎样呢？试试这个：

```
import widget;
import acme.goodies.io; // 多余，但无害

void main() {
   print("Hello");                          // 正确 ...
   widget.print("Hello");                   // ... 正确 ...
   acme.goodies.io.print("Hello");          // ... 也正确!
}
```

对于 io.d 并未造成伤害：widget 定义了一个 acme.goodies.io.print 的别名，这一事实对原有的符号一点都没有影响。额外的那个别名只是到达同一定义的另一种可选方式而已。

最后，你可以看看以前使用 private import 的那段代码。那种用法也是可以接受的，并且等同于普通的 import。

11.1.5 static import 声明

有时，并不是真的想让所有的 import 声明都将导入的模块增加到隐藏的查找列表里（在 11.1.3 节里提及的每种符号查找算法）。有时，只想通过显式符号修饰来访问被某个模块定义的函数（如 modulename.symbolname 对应于 symbolname）。

需要慎重考虑这个问题的最简单情形是：当将一个非常受欢迎的模块与一个具有特殊用途的模块结合在一起使用时，这两个模块间会有一些符号出现重叠。例如，标准模块 std.string 定义了被广泛使用的与字符串相关的例程。如果想与使用了另一种不同编码（如双字节字符集，即 DBCS）的遗留系统进行交互，那么多数时候你会想要使用 std.string 里的符号，而偶尔才会显式地使用你在自己的 dbcs_string 模块里定义的符一号。为实现这一点，只需要在 dbcs_string 的 import 声明里指定 static 即可：

```
import std.string;              // 会定义 string toupper(string)
```

自在读书 www.zizidiary.com

```
static import dbcs_string;      // 也会定义 string toupper(string)

void main() {
    auto s1 = toupper("hello");                 // 正确
    auto s2 = dbcs_string.toupper("hello");     // 正确
}
```

具体来讲，如果上面的代码没有导入 std.string，那么第一个调用就会失败。对于 static import 不会进行自动查找，即使该符号能够明确地解析出来也一样。

static import 在其他场景下也很有用。在一个模块大量导入其他模块时，它可能会想要对自动查找进行控制，并且想使用一种不仅更详细而且更明显的方法。在这种情况下，将 static 与由逗号分隔的列表一起使用则能发挥很大作用：

```
static import teleport, time_travel, warp;
```

将 static 与大括号一起使用也可以达到相同的效果：

```
static {
    import teleport;
    import time_travel, warp;
}
```

11.1.6　选择性导入

另一种解决符号冲突的有效方法是：一个模块只部分导入某些符号。使用下面的语法可以达到这个目的：

```
// 这里是 main.d
import widget : fun, gun;
```

选择性导入就像激光手术一样精确：上面的导入请求只会准确地引入两个符号，即 fun 和 gun。在选择性导入后，甚至连 widget 也是不可见的！假设模块 widget 定义了符号 fun、gun 和 hun。在这种情况下，可以将 fun 和 gun 当做是由 main 定义的这样一种方式使用它们。而其他访问，如 hun、widget.hun 甚至 widget.fun，都是无效的。

```
// 这里是 main.d
import widget : fun, gun;

void main() {
    fun();              // 正确
    gun();              // 正确
    hun();              // 错误!
    widget.fun();       // 错误!
    widget.hun();       // 错误!
}
```

选择性导入的这种高精度和控制使它变得非常受欢迎——大量的程序员对它极其信赖，尤其是那些来自拥有较弱导入和可见性机制的语言的程序员。但需要特别指出的是，D 语言所提供的其他歧义消除机制（前面有所提及），当然也是很有用的。如果 D 语言使用的是一种易于提醒出错的默认查找机制，那么对导入符号的完全控制则会减轻更多负担。

11.1.7　在导入里重新命名

大型项目倾向于创建纷繁复杂的包层次结构。结构过于复杂的目录结构是频繁进行设计造

自在读书 www.zizidiary.com

成的结果，特别是在这样一些设计里，即它们想要建立一个具有包容性、综合性的命名方案，以便让它在面对各种无法预见的项目新增内容时也能保持稳定。因此，像某个模块发现自己正在使用一个深度嵌套的模块这样的情况并不少见：

```
import util.container.finite.linear.list;
```

在这种情况下，重新命名导入变得非常有用，因为它允许你指定一个更短的名字来代替 util.container.finite.linear.list：

```
import list = util.container.finite.linear.list;
```

借助这种 import 声明，程序便可以使用 list.symbol 来代替 util.container.finite.linear.list.symbol。假设正讨论的这个模块也定义了一个类 List，最终的状态如下：

```
import list = util.container.finite.linear.list;

void main() {
   auto lst1 = new list.List;                                    // 正确
   auto lst2 = new util.container.finite.linear.list.List;       // 错误!
      // 符号 util 未定义!
   auto lst3 = new List;                                         // 错误!
      // 符号 List 未定义!
}
```

重新命名导入不会让重新命名后的包（即 util、container……list）可见。因此，就在首次对 util 进行查找时，试图使用那个在 lst2 里定义的冗长的名字会失败。同时，重新命名的导入实际上与静态导入（参见 11.1.5 节）类似，这意味着它不允许自动搜索。而这也正是为什么 new List 不起作用的原因。如果想让这些名字在重新命名后也是可见的，那么使用 alias（参见 7.4 节）则更为方便：

```
import util.container.finite.linear.list;                        // 非静态的
alias util.container.finite.linear.list list;                    // 为了方便

void main() {
   auto lst1 = new list.List;                                    // 正确
   auto lst2 = new util.container.finite.linear.list.List;       // 正确
   auto lst3 = new List;                                         // 正确
}
```

重新命名导入也可以与选择性导入（参见 11.1.6 节）一起使用：

```
import std.stdio : say = writeln;

void main() {
   say("Hello, world!");                  // 正确，调用 writeln
   std.stdio.say("Hello, world");         // 错误!
   writeln("Hello, world!");              // 错误!
   std.stdio.writeln("Hello, world!");    // 错误!
}
```

正如所料，在选择性导入之后，可以对导入的符号重新命名，这样只有最终那个名字是可见的，其他没什么差异。

最后，也可以对模块和导入的符号重新命名：

```
import io = std.stdio : say = writeln, CFile = File;
```

在两个导入更名后的符号之间的可能的相互作用会引发一些矛盾。D 语言的决定仅仅是让上面的声明等同于下面的内容：

自在读书 www.zizidiary.com

```
import io = std.stdio : writeln, File;
import std.stdio : say = writeln, CFile = File;
```

那个双重重命名导入相当于有另外的两个导入操作：一个只将模块重新命名，另一个只对被引入的符号重新命名。这样便能以更为简单的、已知的 import 形式进行语义定义。上面的声明会引入符号 io.writeln、io.File、say 和 CFile。

11.1.8 module 声明

正如 11.1 节所讨论的，渴望成为可导入的 D 语言包和模块，其名字都必须是有效的 D 语言符号，原因很简单：import 只接受这些符号。

还有一些情况，即你会想要让一个模块伪装成与文件名不一样的模块名，以及让包的名字不同于文件实际存放的路径名。最简单的情况是你让模块承载一个并非有效的 D 语言符号的名字的时候。

例如，想要编写一个程序，它遵循一个很大的命名约定，其中包含了-符号，比如 gnome-cool-app.d。那么 D 语言编译器将会拒绝编译它，即使这个程序是完全正确的也一样。这是因为 D 语言在编译过程中必须要为每一个模块生成信息，每一个模块都必须要有一个名字，而 gnome-cool-app 并不是一个有效的名字。一种简单的解决方法是让该源文件换一个不一样的名称，如 gnome_cool_app.d，然后将对最终可执行文件的重新命名作为构建过程的一部分。这种方法绝对可行，但还有一种更简单、更好的方法：在文件的开始部分直接插入一个 module 声明。module 声明看起来有点像这个样子：

```
module gnome_cool_app;
```

如果这个声明存在于 gnome-cool-app.d 里面(同样地,它只能作为文件的第一个声明)，那么编译器就不会有什么问题，因为它会使用名字 gnome_cool_app 来生成所有模块信息。对此，编译器并不会检查实际的名字。甚至可以写成下面这个样子：

```
module path.to.nonexistent.location.app;
```

在这种情况下，编译器将生成所有模块级信息，而这模块就好像叫做 app.d 一样，并且位于 path/to/nonexistent/location 这样一个路径里。编译器并不关心这些，因为它不会对该路径进行定位操作——搜索文件只与 import 相关联，并且在对 gnome-cool-app.d 的直接编译里没有 import。

11.1.9 模块摘要

D 语言构建了这样一种开发模型，即不需要对程序定义的实体进行单独声明（在 C 和 C++ 语言里有“头文件”和“源文件”），你只需将代码放置在一个模块里，然后让其他模块导入这个模块。有时，你也会想采用这样一种开发模型，它要求模块必须实现的签名与这些签名背后的代码能很好地分离开来。在这种情况下，你需要使用根据源代码得到所谓的“模块摘要”（module summary）。模块摘要指的一个模块为使用另一个模块而需要了解的关于它的最少内容。

实际上，模块摘要就是一个没有注释和函数实现的模块。不过对于使用编译时参数的函数的实现，仍然需要将它们保留为模块摘要的一部分。这是因为在编译期间，客户端模块里不可

自在读书 www.zizidiary.com

预见的实例化操作离不开这些带有编译时参数的函数。

模块摘要是有效的 D 语言代码。例如，假设有这样一个模块：

```
/**
这里是本模块的文档注释
*/
module acme.doitall;

/**
这里是类 A 的文档注释
*/
class A {
   void fun() { ... }
   final void gun() { ... }
}

class B(T) {
   void hun() { ... }
}

void foo() {
   ...
}

void bar(int n)(float x) {
   ...
}
```

doitall 的模块摘要会复制这个模块的内容，同时去除所有的注释以及将所有函数体替换为“;”，而对于编译时参数化的函数，则原封不动：

```
module acme.doitall;

class A {
   void fun();
   final void gun();
}

class B(T) {
   void hun() { ... }
}

void foo();

void bar(int n)(float x) {
   ...
}
```

这个摘要包含了某个模块为使用 acme.doital 所需要的信息。多数时候，这个摘要是由编译器内部管理的。在必要时，编译器可以根据源代码（在参考实现 dmd 里是使用-H 标志）生成模块摘要。例如，当想以“头文件加编译库”的形式分发某个库时，生成的摘要就很有用。

不过请注意，函数体的去除操作是没有保证的。出于内联的目的，编译器对非常短的函数体则会保留。例如，如果 acme.doitall.foo 有一个空的函数体或者只是调用了另一个函数，那么其函数体可能会存在于所生成的接口文件里。

这里有一个 C 和 C++编程语言的用户都很熟悉的开发方案，即人为地让头文件（即模块摘要）和实现文件保持分离。虽然这种方法需要做的工作更多，但是它使某些团队管理技术变得更可行。例如，对头文件的写访问可以对设计团队开放，他们控制着这些模块相互提供 API 的

自在读书 www.zizidiary.com

所有细节。实现团队拥有对实现文件的写访问权限，但对头文件只有读访问权限，这些头文件常被用做指导实现的有效文档。编译器会检验实现是否符合接口（至少语法上要符合）。

对于 D 语言，你有 3 种选择：(a) 与模块摘要混在一起使用；(b) 让编译器生成这些模块摘要；(c) 手工管理模块和模块摘要。本书中的所有示例都选择的是 (a) 这一项，让编译器负责关于模块摘要的所有事情。为对其他两种选择进行练习，首先需要在一个类似于图 11-2 的层次结构中排列好各个模块。

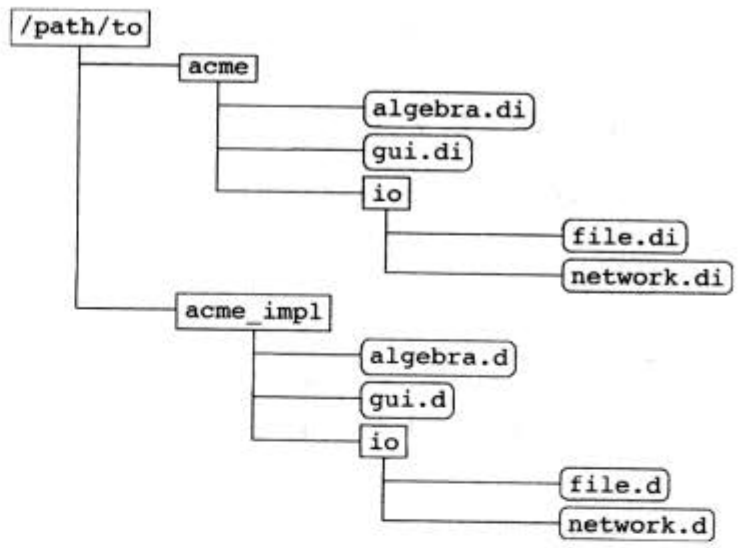


图 11-2　用于将模块概要（“头文件”）与实现文件进行分离的目录结构

使用 acme 的代码需要将 acme 和 acme_impl 的父目录添加到项目的模块搜索根目录里（参见 11.1.2 节），然后使用下面的方法导入模块 acme：

```
// 这里是 client.d
import acme.algebra;
import acme.io.network;
```

目录 acme 只包括了模块概要文件。为获得实现文件的协作，它们的模块名前缀必须加上包名 acme，而不是加上 acme_impl，在这里正好可以使用 module 声明。尽管 algebra.d 位于 acme_impl，但还是可以通过包含下面的声明来声称它实际上是位于 acme：

```
// 这里是 acme_impl/algebra.d
module acme.algebra;
```

相应地，在子包 io 里的模块应该使用：

```
// 这里是 acme_impl/io/file.d
module acme.io.file;
```

这能让编译器生成正确的包名和模块名。为让编译器在构建此程序的过程中能找到函数体，只需将实现文件传递给编译器即可：

```
% dmd client.d /path/to/acme_impl/algebra.d
```

在 client.d 里的 import 指令将会在目录/path/to/acme 里查找接口文件 acme。同时，编译器在显式给定的命令行里会找到那个具有正确包名和模块名的实现文件。

如果 client.d 使用了多个位于 acme 里的模块，那么在编译器的命令行里指定所有这些模块是不可行的。在这种情况下，更好的选择是将 acme 的所有代码都打包进一个二进制库，然后只将这个库传递给 dmd 即可。建立库的语法取决于具体的编译器实现，而在参考实现 dmd 里，可以这样做：

```
% cd /path/to/acme_impl
% dmd -lib -ofacme algebra.d gui.d io/file.d io/network.d
```

开关-lib 会指示编译器构建一个库，而开关-of 输出文件会将输出引向一个名为 acme.lib 的文件（Windows 平台上），或者是名为 acme.a 的文件（各种源于 Unix 的系统）。有了这个库之后，为让客户端代码运行，现在需要做的事情便是执行类似下面的操作：

```
% dmd client.d acme.lib
```

如果 acme 库的应用非常广泛，那么你可以让它成为项目默认库的一部分。这是一种依赖

自在读书 www.zizidiary.com

于实现和系统的操作，需要大家去阅读令人生畏的手册。

11.2　安全性

编程语言的安全性历来都是一个存在争议的概念，但近年来它受到越来越多的关注。

直观地讲，安全的语言指的就是“保护自己的抽象概念”的语言[46，第 1 章]。例如，有一个 D 语言类：

```
class A { int x; }
```

以及一个 D 语言数组：

```
float[] array;
```

D 语言的规则（由语言提供的“抽象概念”）规定：对于所有的 A 类型对象，在更改其成员 x 时不能修改 array 的元素；反之亦然，对于某个 n，在更改 array[n]时不应更改某个 A 类型对象的成员 x。禁止这种无意义的操作看起来是合情合理的，不过与此类似的，在 D 语言里还有一些方法会导致这两种情况发生：通过 cast 来伪造指针，或者使用 union。

```
void main() {
   float[] array = new float[1024];
   auto obj = cast(A) array.ptr;
   ...
}
```

更改 array 里的某个元素（究竟是哪一个与平台相关，通常是第二个或第三个）都会改变 obj.x。

11.2.1　定义行为和未定义行为

在前面展示的示例里，除了那个将指向浮点的指针强制转换成一个类引用的操作有些令人讨厌外，还存在其他的运行时错误——它们完全可以说是由语言未正确履行某些承诺所致。比较典型的错误实例包括空指针解引用、除以零或者提取一个负数的实数平方根。正确的程序不该执行这类操作；然而，事实是在有类型检查的程序里它们仍然可能会发生，这一点可视为是类型系统的失败。

这个针对正确性的“可有可无”准则存在一个问题，即这个名单可能会永远持续增长下去。因此，D 语言将其安全性的概念集中到了一个非常精确和有用的定义上：一个安全的 D 程序指的是它不存在未定义行为。定义行为与未定义行为的区别具体如下。

- 定义行为（defined behavior）：在一个给定状态下，程序段的执行拥有许多确定的输出结果。其中一种可能的结果便是突然结束执行(在解引用空指针或者进行零整除时会发生这种情况)。
- 未定义行为（undefined behavior）：在一个给定状态下，程序段的执行效果是没有定义的。这意味着在现实可能性范畴内什么事情都可能发生。在上一页展示的 cast 便是一个很好的例子——在内部带有这种“癌细胞”的程序很可能只会正常一小段时间，在往 array 写入一些内容后，又紧接着偶然使用 obj 便会导致执行过程失去控制。未检查的数组越界访问、随意的指针运算、悬空指针（参见 4.6 节）——所有这些也都是未定义行为的例子。

自在读书 www.zizidiary.com

（未定义行为非常类似于 Cardelli 引入的“未捕获错误”的概念[15]。他将执行错误分成两大类：捕获的（trapped）和未捕获的（untrapped）。捕获的错误会导致执行立即停止，而未捕获的错误会导致程序执行任意指令。一个带有定义行为的程序绝不可能有未捕获的错误。）

关于定义的和未捕获的行为存在两点很有意思的细微差别。例如，有一门语言，它将整数的“被零除”行为定义为总是产生 int.min。这便使“被零除”成为了定义行为——尽管这不是一种特别有用的定义。其他还有一些类似的处理方法，std.math 实际上将 sqrt(-1) 定义成返回 double.nan。这也是定义行为，因为 double.nan 是一个定义良好的值，它是语言规范和 sqrt 的一部分。实际上“被零除”对于浮点类型并不是一个错误：它被仔细定义来返回正无穷大、负无穷大或者不是一个数（NaN，参见第 2 章）。就 sqrt 或浮点除法而言，程序总会得到可重现的结果。

如果一个程序不会产生未定义行为，那么它就是安全的。

11.2.2　@safe、@trusted 和@system 属性

有一种可用来确保不会发生未捕获的错误的简单方法，即完全禁止所有不安全的 D 语言概念，如 cast 的特定用法。不过，这意味着许多系统都无法使用 D 语言来实现。有时，你必须要跳出某个抽象概念的界限，例如，让看起来拥有某种类型的内存变成另一种不同类型的内存。这正是在内存分配器或垃圾收集器里发生的情况。能够表达这种系统级的软件一直以来都是 D 语言的目标之一。

另一方面，许多应用程序都需要只以一种高度封装的方式来对内存实施不安全的访问。一门语言尽管其垃圾回收器是以一门不安全的语言实现的，但它也可以声称自己是安全的。这是因为从安全语言的角度看，不可能出现对垃圾回收器的不安全使用。垃圾回收器自身被安全语言封装在运行时支持的内部，并且由另一门不一样的语言实现，因此安全语言将它当做是一个具有魔法的原语。在垃圾回收器里缺乏的任何安全性都是实现问题，而非客户端代码的问题。

一个大型项目如何确保其大部分模块的安全性，并且在一些具有选择性的地方仍然服从这些规则呢？针对安全性，D 语言采用的方法是交由用户负责：你可以在声明级别说明代码是否符合安全性或者需要跳出安全性的限制之外。通常情况下，可以在 module 声明后立即指定一个模块特性，如像这样操作：

```
module my_widget;
@safe:
...
```

这时，可以被定义的特性有@safe、@trusted 和@system。这些特性允许模块声明其安全级别。（这种方法并不是什么新鲜内容，Modula-3 就有一种类似的方法，用于区分安全模块和不安全模块。）

遵从@safe 属性的代码承诺只会使用 D 语言的安全子集。也就是说：

- 不会将一个指针类型强制转换至一个非浮点类型（如 int），反之亦然；
- 两个不相关指针类型之间不会进行强制转换；
- 所有数组访问都会进行边界检查；
- 联合不会包含指针类型、类、数组，结构也不会嵌入这种类型；

自在读书 www.zizidiary.com

- 没有指针运算；
- 禁止提取局部变量的地址（事实上所需要的限制是不允许这种地址逃离，因为这更难追踪）；
- 函数调用仅限于其他的@safe 或@trusted 函数；
- 没有内联汇编；
- 没有将 const、immutable 和 shared 强制转换掉；
- 没有使用任何@system 实体。

有时，这些规则可能过于保守。例如，对于指向局部变量的指针，想要避免其逃离便可能将明显合法的程序排除在外。D 语言@safe 子集（称为 SafeD）的力量已相当强大——整个应用程序都可以使用 SafeD 来编写。

正好相反，在一个或一组声明里可以指明：底层访问是有必要的。这种声明应该指定为@system：

```
@system:
void * allocate(size_t size);
void deallocate(void* p);
...
```

@system 属性会有效地关闭所有检查，并且可以毫无约束地驾驭语言的力量——能带来好处也存在风险。

最后，库的责任通常都是向客户端提供安全的抽象概念的位置，而它们自己内部则常会使用一些不安全的特性。D 语言标准库里的许多组件都处于这个位置。这样的声明可以使用@trusted 特性进行标识。

没有指定任何特性的模块会被设定为默认的安全性选项。这个选项可以通过编译器的配置文件和命令行选项进行配置，确切的配置方法则由编译器实现决定。参考实现 dmd 默认是使用@system，而想让@safe 成为默认值，只需要在命令行选项里使用参数“-safe”即可。

在撰写本书时，SafeD 还处在 α 品质阶段（即可能出现不安全程序能编译通过，而安全程序可能无法编译通过的情况），不过它是一个开发非常活跃的领域。

11.3　模块的构造函数与析构函数

有时，模块也需要运行一些初始化代码，它能初始化模块里的一些静态数据。还有一种实现此操作的方法，即在所有访问各个数据的地方插入显式测试（“这个数据要被初始化了吗？”）。当这种操作执行起来有些不太方便或效率低下时，请使用模块的构造函数。

假设有这么一个实例，即要编写某种特定于操作系统的模块，它会根据标志决定具体做什么。在编译期间很容易对主要的操作系统进行区分（例如，“我是 Mac”与“我是 PC”）。但要对不同的 Windows 版本进行区分则只有在运行时才能完成。

为让事情简单些，我们假设代码只关心“Windows Vista 版本”、“比 Vista 新的版本”以及“比 Vista 早的版本”这三者的区别。在模块初始化时进行检测的代码像下面这个样子：

```
private enum WinVersion { preVista, vista, postVista }
private WinVersion winVersion;

static this() {
   OSVERSIONINFOEX info = { OSVERSIONINFOEX.sizeof };
```

自在读书 www.zizidiary.com

```
    GetVersionEx(&info) || assert(false);
    if (info.dwMajorVersion < 6) {
        winVersion = WinVersion.preVista;
    } else if (info.dwMajorVersion == 6 && info.dwMinorVersion == 0) {
        winVersion = WinVersion.vista;
    } else {
        winVersion = WinVersion.postVista ;
    }
}
```

这块代码的突出之处在于模块构造函数 static this()。这种模块构造函数（module constructor）总是在 main 之前执行。一个给定的模块可以包含任意数量的模块构造函数。

相应地，模块析构函数的语法如下：

```
// 在模块级
static ~this() {
    ...
}
```

静态析构函数会在 main 执行完成（通过正常返回或通过异常抛出）后再执行。一个模块可以定义任意数量的模块析构函数，并且可以随意地将模块构造函数和析构函数模块交错放置。

11.3.1　模块内的执行顺序

在一个给定的模块内，其构造函数的执行顺序总是自上而下（词法顺序），模块析构函数的执行顺序则是自下而上（与词法顺序相反）。

如果某个模块构造函数在失败时抛出异常，那么 main 不会被执行，并且只有在词法上位于这个失败的模块构造函数上方的静态析构函数才会被执行。如果有一个模块析构函数失败时抛出异常，那么其他模块的析构函数不会被执行，并且应用程序在终止时会将错误消息输出到标准错误控制台。

11.3.2　跨模块的执行顺序

在跨越不同模块时，顺序的定义变得更加复杂。其规则等同于为静态类构造函数定义的规则（参见 6.3.6 节），并且建立在这样一种概念上，即被其他模块导入的模块一定会优先被初始化，并最后被清理掉。下面是一些规则，可用于决定 moda 和 modb 这两个模块所拥有的静态构造函数的运行顺序：

- moda 和 modb 中至多有一个定义了模块构造函数或模块析构函数。这样就不用担心顺序的问题。
- moda 和 modb 都不会彼此导入。此时的排序未指定——任何顺序都可以工作，因为这两个模块彼此没有依赖。
- moda 导入 modb。那么 modb 的模块构造函数会在 moda 的模块构造函数之前运行，而 modb 模块析构函数会在 moda 的模块析构函数之后运行。
- modb 导入 moda。那么 moda 的模块构造函数会在 modb 的模块构造函数之前运行，而 moda 模块析构函数会在 modb 的模块析构函数之后运行。

自在读书 www.zizidiary.com

- moda 导入 modb，同时 modb 也导入 moda。那么会出现一个错误信号，即“循环依赖”；而执行会在程序加载期间被终止。

目前，对模块循环依赖的检查是在运行时完成的。在编译或连接期间进行检测也是可能的，但意义没那么明显：问题的表象是程序拒绝加载，并且假设程序在传播之前会至少运行一次。不过，早点检测出问题总是好过事后才发现，因此 D 语言把对这种非法情况的检测和报告留给了具体的编译器实现来处理。

11.4　文档注释

文档编写是很无聊的，而无聊是程序员常常抱怨的事。因此，常规的文档通常都很稀少、不完整和过时。

文档自动提取器会尽力根据纯代码推断必要的信息，并显示出两个实体之间的有意义的关系。不过，现今的自动提取器很难记录下具体实现里的高级意图。现代语言通过使用所谓的文档注释（documentation comment）来辅助解决这一问题。这种文档注释就是样式化的注释，其描述的内容包含自定义实体等。语言处理器（要么是编译器本身，要么是一个单独的程序）会结合代码在注释里进行搜寻，并生成某种流行格式（如 XML、HTML 或 PDF）的文档。

D 语言定义了一个文档注释规范，它描述了样式化注释的格式和将它们转换成目标格式的过程。这个过程并不依赖于目标格式，有一个简单且灵活的转换模板（也是由用户定义的）会指导转换器生成几乎任何想要的格式。

对文档注释转换系统进行全面的讲解已超出了本书的范围。这里想说的是，建议你深入了解它。有许多D语言项目，以及编译器参考实现的整个网站和它的标准库都是使用D语言的文档注释生成的。

11.5　与 C 和 C++进行交互

D 语言模块可以直接与 C 和 C++函数进行交互。具体的限制是不允许模板化的 C++函数，因为这需要 D 语言编译器包含一个成熟的 C++编译器。同时，D 语言的类的布局与使用虚拟继承的 C++类是不兼容的。

为调用 C 和 C++里定义的函数，只需在函数声明里指定语言，并确保将你的模块与恰当的库进行连接即可：

```
extern(C) int foo(char*);
extern(C++) double bar(double);
```

这些声明会提示 D 语言编译器使用恰当的栈布局、调用协定和名字编码（即名字碎解[①]）来生成调用，即使在其中的某几个或所有方面 D 语言函数本身会有所不同，也一样会生成。

为了从 C 和 C++内部调用 D 语言函数，只需提供一个实现，其声明与上面类似：

① 名字碎解（name mangling），也叫名字修饰（name decoration）。在现代编程语言里为满足唯一名字解析需要引起的各种问题，而它就是一种用于解决此问题的技术。为将更多的语义信息从编译器传递到连接器，它提供了一种编码方式，以便在函数、结构、类或其他数据类型的名字里加入一些附加信息。参考 http://en.wikipedia.org/wiki/Name_mangling。——译者注

自在读书 www.zizidiary.com

```
extern(C) int foo(char*) {
    ... // 实现
}
extern(C++) double bar(double) {
    ... // 实现
}
```

编译器同样会生成恰当的适合于客户端语言的名字碎解和调用协定。因此该函数便可以从外界进行调用，就像是 D 语言模块一样。

11.6 deprecated 声明

任何声明（类型、函数或数据）的前缀都可以加上 deprecated（弃用的）。这个关键字很像一个存储类别，但它一点都不会影响到实际代码的生成。实际上，它只是告知编译器这个被标识的实体不再建议使用。如果这个实体被用到，那么编译器便会发出警告。如果给编译器传递正确的标志（对于 dmd 来说是-w），那么它实际上会拒绝编译。

通过 deprecated，你可以建立一条缓慢的和有组织的从旧版 API 更新到新版的迁移路径。在恰当地弃用某些声明之后，你可以对编译器进行设置，让 deprecated 声明要么接受，要么拒绝。当这一步准备好时，这些错误便能准确地定位到工作区域，并允许循序渐进地对代码进行更新。

11.7 version 声明

在理想的世界里，你只需要编写一次便可以随处运行。不过在地球上，你偶尔会需要去处理程序里的变化源头——它们有时是库版本，有时是专用构建或者平台依赖性。为满足这种需求，D 语言定义了一个 version（版本）声明，通过它你可以实现有条件地编译代码。

version 的使用相当简单和直接。你可以设置一个版本，也可以对它进行测试。这个版本自身可以是一个整数常量或符号：

```
version = 20100501;
version = FinalRelease;
```

想要测试某个版本，可以这样编写：

```
version(20100501) {
    ... // 声明
}
version (FinalRelease) {
    ... // 声明
} else {
    ... // 更多声明
}
```

如果一个 version 先前被赋有某个值，那么这个测试守护的声明会被编译；否则，它们会被忽略。一个 version 可以有一个 else 子句，其含义不言而喻。

你只能在读取某个版本之前设置它。在某个版本被测试之后，再试图设置它则会产生一个编译时错误：

```
version (ProEdition) {
    ... // 声明
}
version = ProEdition; // 错误!
```

自在读书 www.zizidiary.com

这是因为版本的赋值被假定不是可变的——version 必须相同，无论你的程序中关注的是哪一部分。

除了在源文件内部指定 version 外，也可以在编译器的命令行指定（对于参考实现 dmd，可以使用-version= 123 或-version = xyz）。在命令行和源文件里同时设置 version 也是错误的。

version 的简单性不是偶然的。有很多方法可以轻易地让 version 更强大，但这样做之后，它很快便会背离其初衷。例如，使用#if/#elif/#else 的 C 语言版本处理。在版本控制定义方面，不可否认它提供了更多的可能策略——但这也恰好是为什么会出现这样一种状况的原因，即在典型的 C 语言项目里，版本控制常会拥有一些由条件编译构成的乱成一团的“毛球”。D 语言的 version 被故意设计得没那么强大，其目的在于只支持简单、统一的版本控制。

编译器常会拥有许多预定义的版本，如平台（Win32、Posix 或 Mac）、字节序（LittleEndian、BigEndian）等。version(unittest)会在应用程序启用单元测试功能时被定义。特殊的运行符号__FILE__和__LINE__分别代表的是当前文件的名称和在文件里的行号。完整的 version 定义列表请查询编译器的文档。

11.8　debug 声明

debug（调试）声明是一个特殊的 version，它拥有与之相同的赋值方式和测试语法。特别定义 debug 声明的目的是为了标准化程序定义调试模式和测试设施的方式。

debug 的典型用法如下：

```
module mymodule;
...
void fun() {
    int x;
    ...
    debug(mymodule) writeln("x=", x);
    ...
}
```

如果要调试 mymodule，在编译时可以在命令行定义-debug=mymodule。计算 debug(mymodule)得到的值为 true，进而编译它守护的所有代码。跟 version 一样，在对它进行测试后，不能再将调试符号赋给它。

11.9　D 语言标准库

D 语言的标准库，代号 Phobos，也已随同语言一起有了发展和变化。正因为如此，它既包含了旧式的 API，同时也包含了使用语言新特性的新型库制品。

这个库由两大主要包构成，即 core 和 std。第一个包含的是基础运行时支持，其中包括内建类型的实现、垃圾回收器、启动和关闭代码、并发性支持、用于访问 C 语言运行时库所需要的定义，以及其他相关的组件。std 包含的是更高级的功能。这样组织的好处在于其他库可以建立在 core 之上，并与 std 实现共存，同时不用依赖它。

std 包为一个扁平结构——大部分的模块都直接位于这个包中。这些模块都是致力于特定

自在读书 www.zizidiary.com

的功能领域的。表 11-2 强调和讨论了一些比较重要的 Phobos 模块。

表 11-2　标准模块小结

模 块 名	描　述
std.algorithm	这个模块可以说是 D 语言强大的泛型编程功能的旗舰作品，其灵感源自 C++的标准模板库（STL）[51]。它包含了 70 多个重要的算法，并且是以非常通用的术语实现的。大部分算法都关注于拥有相同元素的结构化序列。在 STL 里，基本的序列抽象概念是迭代器（iterator），而在 D 语言里与之对应的原语是范围（range）。这里的简要说明无法合理表达这个主题，关于 D 语言范围更全面的介绍，请参考在线文档[3]
std.array	用于数组操作的便捷函数
std.bigint	具有高度优化实现的变长整数
std.bitmanip	底层的位操作类型和例程
std.concurrency	与并发性相关的工具，在第 13 章会进行详细讨论
std.container	各种容器的实现
std.complex	复数。出于历史原因，复数是一项内建功能。随着 D 语言和编译器的发展，它正逐渐转向库实现，同时也没有重大的效率损失
std.exception	放置契约相关工具和错误处理工具的地方。值得一提的是 enforce，它在本书中经常用到
std.conv	这里是为满足所有与转换相关的需要而提供全套服务的地方。许多很有用的函数，如 to 和 text 都定义在这里
std.date	与日期和时间有关的操作
std.file	文件实用程序。这个模块通常是将文件作为一个单位进行处理的。例如，函数 std.file.read 会读取整个文件，但 std.file 没有打开一个文件，然后读取一小块的概念（参见下面的 std.stdio）
std.functional	进行函数式定义和编写的原语
std.getopt	命令行解析
std.json	JSON 数据格式处理
std.math	高度优化的数学例程
std.numeric	通用数值算法和核心
std.path	文件路径处理工具
std.random	各种随机数生成器
std.range	与范围相关的定义和分类原语
std.regex	正则表达式引擎
std.stdio	构建于 C 语言的 stdio 库之上的标准 I/O 库工具。文件的输入和输出操作提供的是范围风格的接口，即定义在 std.algorithm 里的许多算法都可以直接对文件进行操作
std.string	特定于字符串的函数。字符串与 std.algorithm 是高度集成的，因此这个模块相对较小，大部分都依赖于（通过定义别名）std.algorithm 中适用于字符串的内容
std.traits	类型特征（trait）和内省（introspection）
std.typecons	定义新类型的工具，如 Tuple
std.utf	处理 UTF 编码的函数
std.variant	可辨识联合类型

自在读书 www.zizidiary.com

第12章

操作符重载

程序员都不太喜欢将内建类型和自定义类型过度区分。给内建类型赋予多种魔法属性与语言的开放性和可扩展性有些背道而驰，因为自定义类型永远都被认定为是第二位的。然而，语言设计者却有充分的理由给内建类型红地毯式的待遇。其中一个原因是，一种语言如果过于可配置便会变得难以教学，以及难以被人和机器解析。每一种语言都试图在内建与可配置之间达到一个良好的平衡，而有一些语言则是试图更靠近两个极端中的某一个。

D语言在这个问题上的看法是务实的：它不仅意识到可配置性的重要性，同时也意识到内建类型的实用性。更具体些，D语言利用内建类型的方式恰好有如下三种。

（1）类型名称语法：数组和关联数组在所有地方都可以使用。现在来看一个实例：与`Array!int`和`AssociativeArray!(string, int)`相比，使用`int[]`和`int[string]`则会更好一些。用户代码无法定义新的用于表达类型名称（如`int[[]]`）的方式。

（2）字面量：数字、字符串、数组和关联数组字面量都是“特殊的”——它们的集合不能被扩展。构造的`struct`对象也是字面量，如`Point(5, 3)`。但一个类型不能定义新的字面量语法，如不能定义`(3, 5)pt`。

（3）语义：编译器会使用某些类型的语义知识以及它们的操作来优化代码。例如，表达式`Hello" ~ ", " ~ "world`在运行时便不会执行任何字符串连接操作，因为编译器知道字符串连接是怎么回事，并且会在编译期间将字符串结合在一起。类似地，编译器会使用算术知识来简化和优化算术表达式。

有些语言将操作符也归到这个列表。它们让操作符变得特殊化：为让自定义类型也可以完成某种运算，必须要使用标准的语言工具，如函数或宏调用。尽管这是一个完全合理的决定，但它却为大量的数值代码带来了麻烦。许多数值程序都会定义它们自己的具有代数运算的类型（无限精度数字、自定义浮点、复数、四元数、八元数、向量、各种布局的矩阵、张量……编程语言无法让它们都成为内建类型）。对于这样的类型，代码的表现力严重不足。与等效的函数式语法相比，操作符通常所需空间更小、所需括号更少，并且会产出更容易识别的代码。例如，要计算三个非零数`x`、`y`和`z`的调和平均数。基于操作符的表达式是非常接近

自在读书
www.zizidiary.com

于数学定义的：

```
m = 3 / (1/x + 1/y + 1/z);
```

在要求使用函数调用而无法使用操作符的编程语言里，情况就没那么好了：

```
m = divide(3, add(add(divide(1, x), divide(1, y)), divide(1, z)));
```

与使用中缀操作符的代码相比，大量包含这种算术操作符的代码很快会变得难以阅读和修改。

D 语言对数值编程来说是一门非常有吸引力的语言。它有可靠的浮点运算和一个优秀的超越函数库（它有时能产生出比系统本地库更高的精度），并且 D 语言还具备复杂建模的能力。这种吸引力被强大的操作符重载工具进一步增强。通过操作符重载，你可以定义自己的数值类型（如用于金融和会计程序的定点数或十进制数、无界整数或无限精度的实数），它可以高度模仿内建数值类型。操作符重载还具备定义拥有类数值代数类型的能力，如向量和矩阵。现在来看看如何定义使用此特性的类型。

12.1　重载操作符

D 语言的操作符重载方法很简单：无论什么时候，只要在某个操作符表达式里至少有一个部分是自定义的类型，那么编译器便会将这个表达式重写成一个常规的带有特定名字的方法调用。然后，应用常规的语言规则。如此看来，重载操作符只不过是方法调用的“语法糖”，因此完全没有必要去学习一个完全成熟的语言特性的各式变化。例如，如果 a 是某种自定义类型，那么表达式 a + 5 会被重写为 a.opBinary!"+"(5)。常规的检查和规则都可应用于方法 opBinary。如果 a 的类型想要支持操作符重载，那么这个类型必须要定义此方法。

重写（更准确地说是“降级”，因为这个过程是将高层概念转换成了底层代码）是一个非常有效的在现有语言特性基础上实现新特性的工具，并且 D 语言大量使用了这种工具。在介绍 scope 语句（参见 3.13 节）的时候，我们已见过降级是如何发挥作用的。在本质上，scope 只是包裹在特意打造的多条 try 语句外层的蜜糖，但是你肯定不会想要直接编写这块降级后的代码——scope 大大提高了讨论的层次。就重写成函数调用而言，操作符重载延续了相同的传统，它定义了所有的操作符调用，同时也利用了常规的函数定义。闲话少说，一起来看看编译器是如何将各类操作符进行降级的。

12.2　重载一元操作符

对于一元操作符“+”（正号）、“-”（负号）、“~”（按位求反）、“*”（指针引用）、“++”（递增）以及“--”（递减），编译器都会将下面这种用于所有自定义类型值的表达式：

```
<op> a
```

重写成

```
a.opUnary!"<op>"()
```

这里的重写其实是一个对 opUnary 方法的调用，在调用时带有一个编译时参数"<*op*>"，同时没有运行时参数。例如，++a 会被重写成 a.opUnary!"++"()。

自在读书 www.zizidiary.com

为了让类型T可以重载多个一元操作符，需要使用下面的签名来定义一个名叫T.opUnary的方法：

```
struct T {
    SomeType opUnary(string op)();
}
```

所有的一元操作符都可以调用上面定义的这个方法。如果想为某些操作符单独定义方法，可以借助于签名约束（参见5.4节）。例如，要定义一个CheckedInt类型，它封装了原始的数字类型，并确保它们的运算绝不会超出边界。CheckedInt应该根据具体要封装的类型进行参数化（如CheckedInt!int、CheckedInt!long等）。CheckedInt带有前增值和前减值操作符的部分定义如下：

```
struct CheckedInt(N) if (isIntegral!N) {
    private N value;
    ref CheckedInt opUnary(string op)() if (op == "++") {
        enforce(value != value.max);
        ++value;
        return this;
    }
    ref CheckedInt opUnary(string op)() if (op == "--") {
        enforce(value != value.min);
        --value;
        return this;
    }
    ...
}
```

12.2.1　使用 mixin 加强操作符定义

有一项非常强大的技术，它不只允许定义一个操作符，而且还允许一次定义一大堆操作符。例如，CheckedInt 的一元操作符“+”、“-”和“~”所做的都是同一件事情——它们都只是将各自的运算转到成员 value。尽管它们是完全不相同的，但可以肯定它们都符合相同的模式。我们可以为每个操作符只定义一个专门的方法，但这会导致毫无意义的样板代码重复。更好的方法是使用字符串 mixin（参见 2.3.4.2 节）直接根据操作数名字和操作符符号装配成这些运算。下面的代码为 CheckedInt 实现了所有相关的一元操作符。

```
struct CheckedInt(N) if (isIntegral!N) {
    private N value;
    this(N value) {
        this.value = value;
    }
    CheckedInt opUnary(string op)()
            if (op == "+" || op == "-" || op == "~") {
        return CheckedInt(mixin(op ~ "value"));
    }
    bool opUnary(string op)() if (op == "!") {
        return !value;
    }
    ref CheckedInt opUnary(string op)() if (op == "++" || op == "--") {
        enum limit = op == "++" ? N.max : N.min;
        enforce(value != limit);
        mixin(op ~ "value;");
        return this;
    }
```

自在读书 www.zizidiary.com

```
    ...
}
```

这在代码规模上的节省很明显的，并且我们在后面遇到二元操作符和索引表达式时，这种情况还会变得更好。此方法的亮点在于 mixin 表达式，它允许你获得一个字符串并要求编译器编译它。这个字符串是通过在字面上人为将操作数和操作符放置在一起而获得的。字符串 op 的这种偶然性用法几乎成为了一种习惯。事实上，整个特性都是在头脑中使用 mixin 来设计的。在以前，D 语言都是为每一种操作符使用一个单独的名称（如 opAdd、opSub、opMul……），这种方法要求强行记住与这些操作符相一致的名字，并且要编写一大堆函数体几乎相同的函数。

12.2.2　后增值与后减值

这里的后增值（a++）和后减值（a--）操作符有点特殊——它们使用了与其对应的前置操作符相同的符号，所以通过符号对它们进行区分是没用的。还有一个挑战是，调用者想获取操作符的结果，那么它必须要能读取这个递增实体的旧值。最后，递增和递减必须与它们的前置和后置版本保持一致。

后增值和后减值完全可以利用一小块的样板代码分别根据前增值和前减值生成。不需要你编写样板代码，D 语言已为你写好了。a++的重写可按如下方法实现（后减值的处理过程类似）。

- 如果不需要 a++的结果，则重写为++a，即重写成 a.opUnary!"++"()。
- 如果需要获得 a++的结果（如 arr[a++]），那么它会被重写成（请深吸一口气）((ref x) {auto t=x; ++x; return t;})(a)。

第一种情况仅仅是承认不使用结果的后增值与前增值是一样的。第二种情况定义了一个 λ 函数（参见 5.6 节），它用于处理所需的样板：为那个输入值创建一个全新的副本，对输入进行递增，然后返回先前那个副本。λ 函数会立即应用到正进行递增的那个值。

12.2.3　重载 cast 操作符

显式强制类型转换是一个拼写为 cast(T)a 的一元操作符。它与其他操作符有点不同：它指定了一个类型，因此它有一个专用的降级处理过程。对于某个值 val，其类型为某种自定义类型，并且要转换成另有一种类型 T，那么

```
cast(T) val
```

会被重写成

```
val.opCast!T()
```

这里的 opCast 实现当然应该返回一个类型为 T 的值，具体细节则由编译器代为执行。虽然不允许函数重载时返回值，但约束模板完全允许多重 opCast 定义。例如，针对某个类型 T，想要定义出由它到 string 和 int 的类型转换，那么它可以编写成下面这个样子：

```
struct T {
    string opCast(T)() if (is(T == string)) {
```

自在读书 www.zizidiary.com

```
        ...
    }
    int opCast(T)() if (is(T == int)) {
        ...
    }
}
```

你可以定义转换到整个类型类别的强制类型转换。以 CheckedInt 示例为基础，定义出到所有内建整数类型的强制转换。其中的诀窍在于某些类型可能有更严格的范围限制，并且我们想通过 enforce 来确保在转换过程中尽量不会出现损失。另一个挑战是，我们同时还想尽量避免使用 enforce，除非真的有必要（如从 CheckedInt!int 到 long 的转换就没必要检查）。因为边界信息在编译期间是存在的，因此我们可以利用 static if（参见 3.4 节）插入必要的强制检查。

```
struct CheckedInt(N) if (isIntegral!N) {
    private N value;
    // 用于所有整数的转换
    N1 opCast(N1)() if (isIntegral!N1) {
        static if (N.min < N1.min) {
            enforce(N1.min <= value);
        }
        static if (N.max > N1.max) {
            enforce(N1.max >= value);
        }
        // 现在就可以安全地进行原始类型转换
        return cast(N1) value;
    }
    ...
}
```

12.2.4　三元操作符重载测试和 if 测试

给定一个自定义类型值，编译器会将

```
a ? ‹expr1› : ‹expr2›
```

重写成

```
cast(bool) a ? ‹expr1› : ‹expr2›
```

类似地，编译器会将 if 语句里的测试

```
if (a) ‹stmt› // 可带也可不带 else 子句
```

重写成

```
if (cast(bool) a) ‹stmt›
```

求反操作符“!”也会被重写为 cast 的否定。

为启用这种测试，需要定义到 bool 的转换，如下面这样：

```
struct MyArray(T) {
    private T[] data;
    bool opCast(T)() if (is(T == bool)) {
        return !data.empty;
    }
    ...
}
```

自在读书 www.zizidiary.com

12.3　重载二元操作符

对于二元操作符+（加）、-（减）、*（乘）、/（除）、%、(求模)、&（按位与）、|（按位或）、^（按位异或）、<<（左移）、>>（右移）、>>>（无符号左移）、~（连接）以及 in（集合隶属测试），表达式（要求 a 和 b 至少有一个为自定义类型）

```
a ‹op› b
```

可以重写为

```
a.opBinary!"‹op›"(b)
```

或者

```
b.opBinaryRight!"‹op›"(a)
```

如果在这些调用里恰好有一个通过了名字查找和重载检查，那么选择的是这个重写版本。如果这两个调用都是有效的，那么会出现一个歧义性错误。如果这些调用都是无效的，那么很明显这是一个“符号未找到”错误。

继续 12.2 节的 CheckedInt 示例，现在来定义所有的二元操作符：

```
struct CheckedInt(N) if (isIntegral!N) {
    private N value;
    // 加法
    CheckedInt opBinary(string op)(CheckedInt rhs) if (op == "+") {
        auto result = value + rhs.value;
        enforce(rhs.value >= 0 ? result >= value : result < value);
        return CheckedInt(result);
    }
    // 减法
    CheckedInt opBinary(string op)(CheckedInt rhs) if (op == "-") {
        auto result = value - rhs.value;
        enforce(rhs.value >= 0 ? result <= value : result > value);
        return CheckedInt(result);
    }
    // 乘法
    CheckedInt opBinary(string op)(CheckedInt rhs) if (op == "*") {
        auto result = value * rhs.value;
        enforce(value && result / value == rhs.value ||
            rhs.value && result / rhs.value == value ||
            result == 0);
        return CheckedInt(result);
    }
    // 除法和余数
    CheckedInt opBinary(string op)(CheckedInt rhs)
            if (op == "/" || op == "%") {
        enforce(rhs.value != 0);
        return CheckedInt(mixin("value" ~ op ~ "rhs.value"));
    }
    // 移位
    CheckedInt opBinary(string op)(CheckedInt rhs)
            if (op == "<<" || op == ">>" || op == ">>>") {
        enforce(rhs.value >= 0 && rhs.value <= N.sizeof * 8);
        return CheckedInt(mixin("value" ~ op ~ "rhs.value"));
    }
    // 按位操作（未检查过，不会溢出）
    CheckedInt opBinary(string op)(CheckedInt rhs)
```

自在读书 www.zizidiary.com

```
            if (op == "&" || op == "|" || op == "^") {
            return CheckedInt(mixin("value" ~ op ~ "rhs.value"));
        }
        ...
    }
```

使用 Intel 处理器的溢出位（这个溢出位由所有的算术运算负责设置或清除），许多检查可能会让性能得到一点提升，但它们却会变得更加依赖于具体的机器。上述代码为每一个特定的操作符测试定义了一个截然不同的操作符。只要有两个及以上的操作符拥有相同的代码外形，它们便会合并在一起。“/”和“%”属于这一情况，因为它们两个做的都是同样的测试；对于所有的移位操作符以及那 3 个按位操作符情况都一样，因为它们并不需要测试。同样地，这个方法就是将运算组合成一个字符串，然后使用 mixin 将字符串编译成一个表达式。

12.3.1　操作符重载的平方

如果操作符重载支持类型定义了它们自己的操作符实现，那么操作符重载的重载（即重载的平方）便可以支持这些类型定义重载操作符的多个重载版本。

例如，表达式 a * 5，其中 a 的类型为 CheckedInt!int。它还无法编译，因为到目前为止 CheckedInt 定义的 opBinary 要求其右侧是一个 CheckedInt。所以客户端代码必须编写成 a * CheckedInt!int(5)才能正常工作，这一点有些不太令人满意。

解决此问题的正确方法是为 CheckedInt!N 定义多个附加的 opBinary 实现。这时，便可以右侧接收类型 N。表面看来，定义新的 opBinary 会有大量重复性的工作，但事实上这只是一行代码的事情：

```
struct CheckedInt(N) if (isIntegral!N) {
    ... // 同上
    // 使用原始数据进行运算
    CheckedInt opBinary(string op)(N rhs) {
        return opBinary!op(CheckedInt(rhs));
    }
}
```

这个方法之所以简单、漂亮，是因为它会将操作符转换成常规的符号，然后再转到另一个操作符实现。

12.3.2　交换律

对于定义操作符的类型出现在一个运算式的右边的情况，如 5 * a，opBinaryRight 的存在是很有必要的。在这种情况下，a 的类型便有机会通过定义 opBinaryRight!"*"(int)对操作符进行挂勾（hooking）操作。这里出现了冗余。例如，如果想在两边都支持与整数一起运算（如 5 * a 和 a * 5），那么需要定义 opBinary!"*"(int)和 opBinaryRight!"*"(int)。这样做很浪费，因为乘法是可交换的。然而，让一门编程语言来决定操作符的交换律可能会存在某些不必要的限制：交换律的特性取决于代数，例如“*”对于矩阵是不能交换的。因此，编

自在读书 www.zizidiary.com

译器让用户自己定义位于左边和右边的操作符，从而在一定程度上减轻了对操作符交换律所负的职责。

当一个操作数毫不费力地转换到另一个类型时，为支持 a ‹*op*› b 和 b ‹*op*› a，只需要下面一行就已足够：

```
struct CheckedInt(N) if (isIntegral!N) {
    … // 同上
    // 实现右侧操作符
    CheckedInt opBinaryRight(string op)(N lhs) {
        return CheckedInt(lhs).opBinary!op(this);
    }
}
```

所有需要我们做的就是获取相应的其左边为 CheckedInt 的表达式。这样，那个定义好的操作符便能发挥作用了。

在某些情况下，进行一次转换需要一些附加的多余步骤。例如，有 5 * c，这里 c 的类型为 Complex!double。前面的解决办法会将乘法运算转到 Complex!double(5) * c，而这个表达式会将 5 转换成一个虚部为零的复数，于是它又会去忙于复数的乘法，而这是没必要的。事实上，这时只用两个实数相乘便足够了。其结果仍然是正确的，只是没必要付出那么多汗水即可得到。对于这种情况，最好的方法是根据交换和非交换运算的不同将右边运算分解开来。交换运算可以简单地通过交换参数实现；而对于非交换运算，则可以根据每一种情况的基础条件，要么一切从头开始实现，要么利用其他已经实现的基础操作来实现。

```
struct Complex(N) if (isFloatingPoint!N) {
    N re, im;
    // 实现可交换性操作符
    Complex opBinaryRight(string op)(N lhs)
        if (op == "+" || op == "*")
    {
        // 假定左边的操作符已实现
       return opBinary!op(lhs);
    }
    // 手工实现非交换性操作符
    Complex opBinaryRight(string op)(N lhs) if (op == "-") {
      return Complex(lhs - re, -im);
    }
    Complex opBinaryRight(string op)(N lhs) if (op == "/") {
        auto norm2 = re * re + im * im;
        enforce(norm2 != 0);
        auto t = lhs / norm2;
        return Complex(re * t, -im * t);
    }
}
```

其他类型可以选择将某些运算组分别组合在一些。在这种情形下，前面针对约束 op 所描述的技术能够提供一些帮助。

12.4　重载比较操作符

对于比较操作符（相等和排序），D 语言遵循的是我们已经在类（参见 6.8.3 节、6.8.4 节）那一章

自在读书 www.zizidiary.com

遇到过的设计。这主要还是一个历史原因形成的决定，不过想要追求一种与普通 opBinary 不一样的比较设计，也有很多很不错的其他理由。首先，在==和!=之间，以及<、<=、>、>=这四者之间都有着很强的联系。这种联系暗示着使用两个命名函数会比支持基于符号的分别定义每个操作符的代码更好。另外，与这一整套的操作符相比，还会有更多的类型可能定义相等性和排序。鉴于这一事实，D 语言提供了一个小巧、简单的工具来定义比较运算，而没有强制使用功能强大的 opBinary 工具。

a == b 的重写会按照下面的算法进行，这里的 a 和 b 至少有一个为自定义类型。

- 如果 a 和 b 都是类类型，那么这个重写为 object.opEquals(a, b)。正如 6.8.3 节所描述的，类之间的比较会得到一个小协议，它被实现在核心库的 object 模块里。
- 否则，如果 a.opEquals(b)和 b.opEquals(a)会解析成相同的函数，那么这个重写为 a.opEquals(b)。当 a 和 b 拥有相同的类型，不管修饰符相同还是不同，都会发生这种情况。
- 否则，a.opEquals(b)和 b.opEquals(a)中恰好有一个一定是可编译的，并且会成为最终的重写。

使用“<”、“<=”、“>”、“>=”这 4 个排序比较操作符的表达式会按下面的算法重写。

- 如果a.opCmp(b)和b.opCmp(a)都会解析成相同的函数，那么这个重写为a.opCmp(b) ‹*op*› 0。
- 否则，a.opEquals(b)和 b.opEquals(a)中恰好有一个一定是可编译的。如果第一个可编译，那么这个重写是 a.opCmp(b) ‹*op*› 0；否则，重写为 0 ‹*op*› b.opCmp(a)。

再次值得一提的是，opEquals 和 opCmp 两者都存在的原因。乍一看，似乎只要 opCmp 就足够了（对于相等运算，可以使用 a.opCmp(b)== 0）。然而，尽管大多数类型可以定义相等运算，但许多类型无法轻松地定义不相等关系。例如，矩阵或复数拥有相等运算，但它们缺少“正则有序”（canonical ordering）关系。

12.5　重载赋值操作符

赋值操作符包括 a = b，以及像 a += b 和 a *= b 这样的原址二元操作符（in-place binary operator）。在 7.1.5.1 节，我们已知道：

```
a = b
```

可以被改写成

```
a.opAssign(b)
```

对于原址二元操作符，表达式

```
a ‹op›= b
```

会被重写成

```
a.opOpAssign!"‹op›"(b)
```

这种重写让 a 的类型有机会遵循上面描述的技术实现原址运算。假如，想要为 CheckedInt 实现+=：

自在读书 www.zizidiary.com

```
struct CheckedInt(N) if (isIntegral!N) {
    private N value;
    ref CheckedInt opOpAssign(string op)(CheckedInt rhs)
          if (op == "+") {
        auto result = value + rhs.value;
        enforce(rhs.value >= 0 ? result >= value : result <= value);
        value = result;
        return this;
    }
    ...
}
```

在上面的定义里有三点细节值得注意。第一点，opOpAssign 会返回一个对当前对象的引用，这样它便可以与内建类型的工作方式相兼容。第二点，实际的计算不会原址完成，而是偷偷进行，此对象实际的状态只有在验证通过之后才能被改变；否则，存在损坏当前对象的风险，到那时 enforce 会抛出异常。第三点，这个操作符的主体部分实际上是重复 12.3 节的 opBinary!"+"的内容。在原址赋值操作符的定义里，我们可以使用最后一个发现来衡量已存在的所有二元操作符的实现。具体定义如下：

```
struct CheckedInt(N) if (isIntegral!N) {
    ... // 同上
    // 定义所有的赋值操作符
    ref CheckedInt opOpAssign(string op)(CheckedInt rhs) {
        value = opBinary!op(rhs).value;
        return this;
    }
}
```

另外，一个类型如果涉及重新定义赋值操作符，那么它可以选择通过定义二元操作符来实现。考虑到效率时，可能做出这种决定。对许多类型来说，可以原址修改对象能占用更少的空间，并且比创建新的对象速度更快。

12.6　重载索引操作符

D 语言支持高效地定义完全抽象的数组：它支持一般情况下期望一个数组所应具备的所有操作，但它绝不会将其元素的地址暴露给客户端代码。索引操作符的重载是这一能力的重要贡献者。为达到正确的索引访问效果，编译器将元素的读取和写入进行了区分。元素写入需要将一个数组元素作为赋值操作符的左边，无论是普通的=，还是原址二元操作符（如+=）都一样。

当不是进行赋值时，对于任意的参数 k，编译器会将表达式

```
a[b1, b2, ..., bk]
```

重写成

```
a.opIndex(b1, b2, ..., bk)
```

opIndex 方法的实现需要决定像“可以接受多少个参数”，“它们的类型应该是什么”，以及“返回类型是什么”这样一些内容。

如果一个索引操作符的结果被赋值，那么表达式

```
a[b1, b2, ..., bk] = c
```

会变换成

自在读书 www.zizidiary.com

```
a.opIndexAssign(c, b1, b2,..., bk)
```

如果在一个索引表达式里使用了增值或减值运算，那么表达式（这里的‹*op*›为++或--）

```
‹op› a[b1, b2,..., bk]
```

会重写为

```
a.opIndexUnary!"‹op›"(b1, b2,..., bk)
```

后增值和后减值会根据前增值和前减值自动生成，与 12.2.2 节描述的内容一样。

最后，如果某个索引的元素被原址修改，那么表达式

```
a[b1, b2,..., bk] ‹op›= c
```

会变换成

```
a.opIndexOpAssign!"‹op›"(c, b1, b2,..., bk)
```

这些重写让 a 的类型能够完整地定义索引元素是如何被访问和操作的。为什么索引类型要对赋值操作符负责呢？只是让 opIndex 返回一个对存储元素的引用，这样的设计显然更好一些。例如：

```
struct MyArray(T) {
    ref T opIndex(uint i) {...}
}
```

这样，无论 T 支持的是什么样的赋值和修改赋值操作，它们都可以正常工作。例如，有一个类型为 MyArray!int 的数组 a，则表达式 a[7] *= 2 首先会从 opIndex 获得 ref int，然后将该引用就地乘以 2。事实上，内建数组就是这样工作的。

不过，这个简单设计存在缺陷。其中一个问题是，许多类似数组一样的集合并不想通过 ref 访问它们的成员。如果可能的话，它们希望能对其成员的位置进行封装和抽象。这样做的好处正是信息隐藏常见的优势——容器可以有更多的自由为其元素选择最好的存储策略。有一个简单示例，即定义一个存储 bool 对象的容器。如果这个容器被迫允许访问 ref bool，那么它必须要在一个单独的地址存储每一个值。如果这个容器可以将这些地址隐藏掉，那么它可以在 1 个字节里存储 8 个 bool 值。

另外一个示例是，对于某些容器，数据访问与数据操作之间是无法分辨的。例如，稀疏数组。稀疏数组在表面上可能有数百万个元素，但实际上只有少数几个非零值。这时，可以让稀疏数组采用空间效率比较高的存储策略。现在来看下面的代码：

```
SparseArray!double a;
...
a[8] += 2;
```

数组该做什么，除了取决于新数据外，还取决于当前数组在这一时刻的内容：如果 a[8] 槽之前为空，那么创建此槽，并将其值置为 2；如果此槽的值为-2，那么将此槽移除，因为新值为零，这个是不需要显式存储的；如果此槽包含的是除-2 以外的其他值，那么使用加法运算，并将结果存储回槽里。如果要求 opIndex 必须返回一个引用，则无法实现这样的效果，即使其中能带来好处的部分也无法达到。

12.7　重载分割操作符

D 数组提供了分割操作符 a[]和 a[b1..b2]（参见 4.1.3 节）。这两种操作符也都可以被

自在读书 www.zizidiary.com

自定义类型重载。编译器会执行类似于索引操作符那样的降级操作。

当不涉及赋值运算时，编译器会将 a[]重写成 a.opSlice()，而 a[b1..b2]会重写成 a.opSlice(b1, b2)。

分割操作符的降级遵循的模具与为数组定义的模具相同。方法名都使用 slice 代替 index，其中‹op› a[]会降级成 a.opSliceUnary! "‹op›"()，而‹op› a[b1..b2]会变成 a.opSliceUnary!"‹op›"(b1,b2)，a[] = c 会变成 a.opSliceAssign(c)，a[b1 .. b2] = c 会变成 a.opSliceAssign(c, b1, b2)，a[] ‹op›= c 会变成 a.opSliceOpAssign!"‹op›"(c)，最后 a[b1..b2] ‹op›= c 会变成 a.opSliceOpAssign! "‹op›"(c, b1, b2)。

12.8　$操作符

在内建数组的索引和分割表达式的内部，D 语言允许使用符号“$”来代表数组长度。例如，a[0 .. $-1]会选择内建数组 a 除最后一个元素以外的所有元素。

虽然这看起来是个很小的功能，但“$”已让大家享受到使用 D 语言数组的乐趣。与之相反的是，“$”既“神奇”又无法被重载的事实长久以来已成为一块心病——有证据显示内建类型是有能力访问自定义类型的。

对于自定义类型，可以像下面这样重载操作符“$”

- 在表达式 a[‹*expr*›]里，其中 a 是一个自定义类型值，如果“$”出现在表达式‹*expr*›里，那么它可以重写为 a.opDollar()。不管此表达式是否被赋值，重写都是一样的。
- 在表达式 a[‹$expr_1$›, . . . , ‹$expr_k$›]里，如果“$”出现在‹*expri*›里，那么它会被重写为 a.opDollar!(i)()。
- 在表达式 a[‹$expr_1$› .. ‹$expr_2$›]里，如果“$”在‹$expr_1$›或者‹$expr_2$›里，那么它会重写为 a.opDollar()。

如果 a 是某个表达式的结果，那么这个表达式只会计算一次。

12.9　重载 foreach

自定义类型有能力从本质上定义 foreach 如何与它们协同工作。对集合、范围、流及其他可被迭代的实体这样一些模型化的类型来说，这是一个天大的好消息。然而更妙的是，还有两种截然不同的方法供你在需要时进行取舍。

12.9.1　带有迭代原语的 foreach

有一种方式可以定义 foreach 如何与你的类型（struct 或 class）一起工作，即定义三个迭代原语：类型为 bool 的 empty 特性用于判别是否还剩余元素，特性 front 返回正被

自在读书 www.zizidiary.com

迭代的当前元素，而方法 popFront() 则移动到下一个元素。这三个原语的典型实现如下所示：

```
struct SimpleList(T) {
private:
    struct Node {
        T _payload;
        Node * _next;
    }
    Node * _root;
public:
    @property bool empty() { return !_root; }
    @property ref T front() { return _root._payload; }
    void popFront() { _root = _root._next; }
    ...
}
```

有了这个定义，迭代一个列表现在便是一件很简单的事了：

```
void process(SimpleList!int lst) {
    foreach (value; lst) {
        ... // 使用类型为 int 的值
    }
}
```

编译器会将 foreach 控制的代码重写成更笨拙但更精细的使用这三个原语实现的 for 循环：

```
void process(SimpleList!int lst) {
    for (auto __c = lst; ! __c.empty; __c.popFront()) {
        auto value = __c.front;
        ... // 使用类型为 int 的值
    }
}
```

如果为 value 指定 ref，那么编译器会在整个循环体内将所有 value 的使用替换成对 __c.front 的调用。这样，你便可以直接替换列表中的元素。当然，front 特性必须要返回对自身的 ref；否则，在将它用做左值时会出现错误。

最后还有一点，如果被迭代对象提供不带参数的分割操作符 lst[]，那么__c 会被初始化成 lst[] 而不是 lst。这是为了在从一个容器里“提取”迭代方式时，无需要求此容器定义那三个迭代原语。

12.9.2　带有内部迭代的 foreach

上面的原语对外暴露一个迭代接口，客户端代码可以按自己想要的方式来使用它。有时，使用“内部迭代”会更好些。也就是说，被迭代实体需要完全控制迭代的过程，并且自己执行循环体。这种反向控制在许多情况下都能派上用场，特别是在递归遍历集合（如树）时最适合。

为了达到内部迭代与 foreach 一起使用的效果，需要为结构或类定义一个 opApply 方法。例如：

```
import std.stdio;

class SimpleTree(T) {
private:
    T _payload;
    SimpleTree _left, _right;

public:
    this(T payload) {
        _payload = payload;
```

自在读书 www.zizidiary.com

```
    }

    // 树的中序遍历
    int opApply(int delegate(ref T) dg) {
        auto result = dg(_payload);
        if (result) return result;
        if (_left) {
            result = _left.opApply(dg);
            if (result) return result;
        }
        if (_right) {
            result = _right.opApply(dg);
            if (result) return result;
        }
        return 0;
    }
}

void main() {
    auto obj = new SimpleTree!int(1);
    obj._left = new SimpleTree!int(5);
    obj._right = new SimpleTree!int(42);
    obj._right._left = new SimpleTree!int(50);
    obj._right._right = new SimpleTree!int(100);
    foreach (i; obj) {
        writeln(i);
    }
}
```

这个程序实现了树的前序遍历，并且会输出：

```
1
5
42
50
100
```

编译器会将循环体（在本例里为{ writeln(i); }）包裹成一个委托（delegate），并将它传递给 opApply。编译器会将所有事情都安排好，这样提前跳出此循环的代码会在退出委托时返回 1，从而操纵 opApply 内的 result。

借助这些信息，opApply 读起来容易多了：先将循环体应用到根节点，然后递归左、右节点。实现的简洁性是关键，真是这样。如果尝试使用 empty、front 和 popFront 来实现树的迭代，那么事情将变得异常复杂。这是因为在 opApply 里的迭代状态在调用栈上是被隐式维护的。如果使用这三个迭代原语，你需要显式地维护这个状态。

在 opApply 和 foreach 之间的相互作用里，还有一个细节值得注意。在 foreach 里使用的那个变量 i 是 delegate 类型的一部分。幸运的是，这个变量的类型，甚至在 foreach 里使用的那些绑定参数的数量都是完全可配置的。如果将 opApply 定义成只接收一个具有两个参数的委托，那么可以像下面这样使用 foreach：

```
foreach (k, v; object) {
    ...
}
```

事实上，用于内建关联数组的键和值的迭代便是使用 opApply 实现的。对于一个类型为 V[K] 的关联数组，opApply 接受的委托会将 K 和 ref V 用做参数。

自在读书 www.zizidiary.com

12.10　在类里定义重载操作符

前面大多数的重写操作使用的都是如 opBinary(string)(T)那样带有编译时参数的方法。这种方法除了在结构内部可以使用外，在类的内部也同样可以使用。唯一的问题是，这个带有编译时参数的方法隐式为 final，并且不能被改写。因此，在定义一个带有可改写成员的类或接口时，必须执行一些额外的步骤。最简单的解决方案是让如 opBinary 这样的方法转到一个可被改写的常规方法：

```
class A {
    // 不可改写方法
    A opBinary(string op)(A rhs) {
        // 转到一个可改写的函数
        return opBinary(op, rhs);
    }
    // 可重载的方法，在运行时会处理字符串
    A opBinary(string op, A rhs) {
        switch (op) {
            case "+":
                ... // 实现加法
                break;
            case "-":
                ... // 实现减法
                break;
            ...
        }
    }
}
```

这种方法很有效，但不是最优的。因为在运行期间它要对操作符进行测试，而这是一个可以在编译过程中执行的动作。下面的解决方案可以消除这个效率低下的问题，方法是将这个测试移到泛型的 opBinary 内部：

```
class A {
    // 不可改写方法
    A opBinary(string op)(A rhs) {
        // 转到一个可改写的函数
        static if (op == "+") {
            return opAdd(rhs);
        } else static if (op == "-") {
            return opSubtract(rhs);
        } ...
    }
    // 可改写的方法
    A opAdd(A rhs) {
        ... // 实现加法
    }
    A opSubtract(A rhs) {
        ... // 实现减法
    }
    ...
}
```

这次，每个操作符都有了一个可改写的方法。当然，也可以选择重载这些操作符，以及选择按需要将它们分组的方式。

自在读书 www.zizidiary.com

12.11　完全不一样的重载：opDispatch

也许最有意思且最具特色的重写是 opDispatch，它将 D 语言带到了通常是专属于大量动态语言的地方。

如果某种类型 T 定义了方法 opDispatch，那么对于所有 a 的类型未曾定义的可能为 fun 的方法（未定义时，调用它会导致“方法未定义”错误），编译器则会将

```
a.fun(<arg1>, ..., <argk>)
```

重写成

```
a.opDispatch!"fun"(<arg1>, ..., <argk>)
```

opDispatch 的定义可以实现大量非常有意思的、活力十足的设计。例如，有一个 opDispatch 实现，它要为某个类的方法实现另一种命名协定。首先，一起来实现一个简单的函数。这个函数可以将符号 written_like_this 的各个部分转换成“骆峰样式”，即 writtenLikeThis：

```
import std.ctype;

string underscoresToCamelCase(string sym) {
    string result;
    bool makeUpper;
    foreach (c; sym) {
        if (c == '_') {
            makeUpper = true;
        } else {
            if (makeUpper) {
                result ~= toupper(c);
                makeUpper = false;
            } else {
                result ~= c;
            }
        }
    }
    return result;
}

unittest {
    assert(underscoresToCamelCase("hello_world") == "helloWorld");
    assert(underscoresToCamelCase("_a") == "A");
    assert(underscoresToCamelCase("abc") == "abc");
    assert(underscoresToCamelCase("a_bc_d_") == "aBcD");
}
```

有了 underscoresToCamelCase，我们便能轻易地为类定义 opDispatch 方法，让它接受 a.method_like_this()这种形式的调用，并且自动地将这些调用转到 a.methodLikeThis()——这一切都是在编译期间进行的。

```
class A {
    auto opDispatch(string m, Args...)(Args args) {
        return mixin("this."~underscoresToCamelCase(m)~"(args)");
    }
    int doSomethingCool(int x, int y) {
        ...
        return 0;
    }
```

自在读书 www.zizidiary.com

```
}

unittest {
    auto a = new A;
    a.doSomethingCool(5, 6);     // 直接调用
    a.do_something_cool(5, 6);   // 通过 opDispatch 进行同样的调用
}
```

第二个调用没有指向 A 的任何方法，因此它会通过调用 a.opDispatch!"do_something_cool"(5, 6) 路由到 opDispatch。接着，opDispatch 会生成字符串 "this.doSomethingCool(args)"，然后使用 mixin 来编译它。由于 args 已绑定到参数对“5, 6”，因此 mixin 最终会转到 a.doSomethingCool(5, 6)——正是之前那个漂亮的转向操作。至此，任务完成！

使用 opDispatch 进行动态分派

尽管使用 opDispatch 在编译时耍出各种花招很有意思，但真正有意思的是应用程序由此具有了动态性。动态语言（如 JavaScript 或 Smalltalk）支持运行时给对象添加方法。在 D 语言里，我们可以定义一个 Dynamic 类（它支持动态地添加、移除及调用方法），并尝试实现类似的功能。

首先，我们需要为这种动态方法定义一个运行时签名。在这里，定义在模块 std.variant 里的“百变”类型 Variant 能提供不少的帮助。一个 Variant 对象可容纳任何值，这让它成为理想的候选类型，用于动态方法的参数和返回类型。这样，我们将这种动态方法的签名定义成一个 delegate，它会将 Dynamic 作为第一个参数（填补 this 的角色），一个 Variant 数组作为其他参数，并会返回一个 Variant。

```
import std.variant;

alias Variant delegate(Dynamic self, Variant[] args...) DynMethod;
```

因为有了“...”，所以我们可以使用任意数量的参数来调用 DynMethod，并且让编译器将它们包裹成一个数组。现在来定义 Dynamic。按照承诺，它允许在运行时对方法进行处理。为此，Dynamic 需要定义一个关联数组，此数组会将 string 映射到 DynMethod：

```
class Dynamic {
    private DynMethod[string] methods;
    void addMethod(string name, DynMethod m) {
        methods[name] = m;
    }
    void removeMethod(string name) {
        methods.remove(name);
    }
    // 动态分派方法
    Variant call(string methodName, Variant[] args...) {
        return methods[methodName](this, args);
    }
    // 使用 opDispatch 来提供语法糖
    Variant opDispatch(string m, Args...)(Args args) {
        Variant[] packedArgs = new Variant[args.length];
        foreach (i, arg; args) {
            packedArgs[i] = Variant(arg);
        }
```

自在读书 www.zizidiary.com

```
            return call(m, args);
        }
    }
```

对 Dynamic 进行测试：

```
unittest {
    auto obj = new Dynamic;
    obj.addMethod("sayHello",
        delegate Variant(Dynamic, Variant[]args...) {
            writeln("Hello, world!");
            return Variant();
        });
    obj.sayHello(); // 输出"Hello, world!"
}
```

添加一个方法会造成一定数量的“语法糠”[①]（syntactic chaff），因为所有方法都必须符合同样的签名。在上面的示例里，有许多未用到的内容。这是因为新增加的委托不会使用任何参数，也不会返回有意义的值。不过，这种调用语法非常简洁，这才是最重要的。因为通常情况下很少会去添加方法，调用才是很平常的事。有多种方式可以对 Dynamic 进行改善，例如，定义一个 getMethodInfo(string) 检查函数，让它返回参数的数量和指定方法的类型。

值得注意的是，对要静态地还是要动态地去做某些事情这两种方式的抉择，需要仔细地权衡。在运行时你做得越多，越是需要符合公共的数据格式（即 Variant），并且在效率上做出妥协（如在运行时对方法名进行查找）。其回报是增加了灵活性——你可以在运行时根据所需操纵类的定义，其中包括定义动态继承关系、与脚本语言进行交互、为你自己的对象定义脚本等诸多内容。

12.12　小结与快速参考

自定义类型可以重载大部分的操作符。当然也有例外，如逗号操作符“,”、逻辑与操作符“&&”、逻辑或操作符“||”、一致性测试“is”、三元操作符“?:”、一元操作符取地址“&”及 typeid。如果对这些操作符进行重载，则造成的困惑远大于带来的灵活性。

说到困惑，操作符重载就是一种强大而又需要小心使用的工具。在 D 语言里，建议最好不要将操作符重载用于某些比较怪异的意图，如定义完整的特有领域嵌入式语言（domain-specific embedded language，DSEL）。如果想定义 DSEL，最好使用字符串、字符串 mixin（参见 2.3.4.2 节），以及编译时函数计算（参见 5.12 节）来解析从编译时字符串输入的 DSEL，然后生成相应的 D 语言代码。这需要大量的工作，但库的客户端会从中获得好处。

opDispatch 的定义开拓了新视野，但在使用它时要负有责任感。过于动态化可能会引入不必要的低效率，并且削弱类型检查——请记住这一点：如果在上面的示例里，你写的是 a.heloWorld() 而不是 a.helloWorld()，那么这种代码仍可以编译，只有在运行时它才会提示失败。

表 12-1 总结了本章所提供的信息。在为自己的类型重载操作符时，可以将它用做一张备忘表。

① 即无意义的东西。——译者注

自在读书 www.zizidiary.com

表 12-1　重载操作符

表达式	重写结果
‹*op*›a，其中‹*op*› ∈ {+,-,~,*,++,--}	a.opUnary!"‹*op*›"()
a++	((ref x) {auto t=x; ++x; return t;})(a)
a--	((ref x) {auto t=x; --x; return t;})(a)
cast(T) a	a.opCast!(T)()
a ? ‹$expr_1$› : ‹$expr_2$›	cast(bool)a ? ‹$expr_1$› : ‹$expr_2$›
if (a) ‹*stmt*›	if (cast(bool)a) ‹stmt›
a‹*op*›b，其中‹*op*› ∈ {+,-,*,/,%,&,\|,^,<<,>>,>>>,~,in}	a.opBinary!"‹*op*›"(b)或者b.opBinaryRight!"‹op›"(a)
a == b	如果a、b为类：object.opEquals(a, b)（参见6.8.3节）。否则，如果a和b类型相同：a.opEquals(b)。否则，为a.opEquals(b)和b.opEquals(a)之间唯一可编译那个
a != b	!(a == b)，然后参考上面的重写
a < b	a.opCmp(b) < 0 或者 b.opCmp(a) > 0
a <= b	a.opCmp(b) <= 0 或者 b.opCmp(a) >= 0
a > b	a.opCmp(b) > 0 或者 b.opCmp(a) < 0
a >= b	a.opCmp(b) >= 0 或者 b.opCmp(a) <= 0
a = b	a.opAssign(b)
a ‹*op*›= b，其中‹*op*› ∈ {+,-,*,/,%,&,\|,^,<<,>>,>>>,~}	a.opOpAssign!"‹*op*›"(b)
a[b_1, b_2, ..., b_k]	a.opIndex(b_1, b_2, ..., b_k)
a[b_1, b_2, ..., b_k] = c	a.opIndexAssign(c, b_1, b_2, ..., b_k)
‹*op*›a[b_1, b_2, ..., b_k]，如果‹*op*› ∈ {++, --}	a.opIndexUnary(b_1, b_2, ..., b_k)
a[b_1, b_2, ..., b_k] ‹*op*›= c，其中‹*op*› ∈ {+,-,*,/,%,&,\|,^,<<,>>,>>>,~}	a.opIndexOpAssign!"‹*op*›"(c, b_1, b_2, ..., b_k)
a[b_1 .. b_2]	a.opSlice(b_1 .. b_2)
‹*op*›a[b_1 .. b_2]	a.opSliceUnary!"‹*op*›"(b_1, b_2)
a[] = c	a.opSliceAssign(c)
a[b_1 .. b_2] = c	a.opSliceAssign(c, b_1, b_2)
a[] ‹*op*›= c	a.opSliceOpAssign!"‹*op*›"(c)
a[b_1 .. b_2] ‹*op*›= c	a.opSliceOpAssign!"‹*op*›"(c, b_1, b_2)

自在读书 www.zizidiary.com

第 13 章

并发性

在硬件行业，各种因素的汇集已经导致我们在访问计算资源的方式上发生了质的变化，这种变化又进一步促使我们在处理计算的方式上，以及在使用的语言抽象概念方面都进行了深远变革。并发性现在几乎无处不在，利用好它是软件的责任。

虽然软件行业作为一个整体对并发性革命带来的挑战还没有最好的回应，但 D 语言的年轻让其缔造者可以就并发性做出明智的决定，同时不用被过去废弃的选择或大量遗留代码库所束缚。在打造成并发命令式语言方面，它已有了重大突破，即 D 语言不会产生线程间的数据共享。默认情况下，并发线程实际上是被语言机制孤立的。允许数据共享，但它是以有限制的、可控的方式实现的，这让编译器有能力提供具有说服力的全局性担保。

同时，D 语言本质上仍然是一门系统编程语言，因此它允许将各种很底层的、独特的方法用于并发性。只不过，其中一些机制不允许在安全程序里使用。

简言之，这里列出了 D 语言是如何提供并发性支持的：

- 并发性的主要方法是使用孤立的线程或进程，它们之间再通过消息进行通信。这种范型被称为“消息传递”（message passing），它能让程序变得安全和模块化，而这种程序也很容易理解和维护。有各式各样的语言和库成功应用了消息传递。在历史上，消息传递方法比基于内存共享的方法在速度上要慢一些，这也正是它为什么没被完全采用的原因。不过近年来，这一趋势已经历了一个明显的逆转，并且这一趋势还在持续。鼓励大家的 D 语言并发程序都使用消息传递，这是一种得益于大量基础设施支持的范型。
- D 语言还提供了对旧式的基于临界区（通过互斥体和事件变量实现保护）的同步支持。这种并发方法近年来遭到了严厉的批评，因为它未能很好地扩展到当今和将来都是高度并行的架构里去。D 语言推行的是对数据共享进行严格的控制，因此它对基于锁的编程风格进行了限制。这些限制在一开始看起来可能相当严厉，但它们解决了基于锁的代码所存在的最大难题：底层数据竞争。不过，数据共享依然被保留，它是跨线程传递大量数据的最有效的方式，因此它不应被忽视。

自在读书
www.zizidiary.com

- 遵照系统级语言的传统，未标记为@safe 的 D 语言程序可以通过强制类型转换来获得烫手的、泡沫横飞的、未经检查的数据共享。这些程序的正确性很大程度上便要由你来负责。
- 如果这种级别的控制还无法满足你的需要，你还可以使用 asm 语句来对计算机资源进行终极控制。如果还想要进行比这更低级别的控制，那么你可能需要一把微型烙铁和一只非常、非常稳定的手。

在进入这些厚重的话题之前，先卖个关子，其目的是要更好地理解一下撼动这个世界的硬件发展。

13.1　并发门

当谈及并发性时，我们正生活在一个超越以往任何时期的广受关注的年代。这种年代的外观形式混合了好的和坏的消息，这些消息导致了一种复杂的景象：权衡、强制和趋势。

好消息是，集成度仍在以摩尔定律继续增加。据我们所了解到的，以及目前我们所能够合理规划的，这一趋势至少在撰写本书之后的十年内会持续下去。小型化的增加导致了计算功率密度的增加，因为在单位面积的单元里需要将更多的晶体管放置在一起工作。由于组件靠得更紧密，其连接也变得更短，这意味着局部互连会变得更快。这可是效率之源。

遗憾的是，有许多语句从一开始就已“不幸”，它们严重束缚了在增加计算密度方面的热情。举例来说，连接性不仅仅是局部的——其组成是一个层次结构[16]：由紧密连接的组件构成的单元一定会连接到其他单元，从而组成更大的单元。以此下去，较大的单元也会连接到其他较大的单元，从而组成更大的功能块，依次类推。在连接性方面，这种更大的功能块彼此间会保持“很远的距离”。更糟的是，由于这些功能块之间的连接是通过减少连线的厚度以及它们之间的距离实现的，因此随着每一功能块复杂性的增加，它们之间连接的复杂性也会增加。这意味着会增加电阻、电容和串扰。电阻和电容会让导线的传播速度变得很槽。串扰指的是在某根导线通过电磁场（在本例里）传播至邻近导线时，这根导线里所出现的信号偏置。在高频段，导线就是一根天线，而串扰会变得让人无法忍受，于是串行通信逐渐取代了并行通信。这种有点违背常理的现象随处可见——USB 取代了并口，SATA 取代了 PATA 成为磁盘数据连接器，在内存子系统里串行总线也正在取代并行总线，所有这些都要归咎于串扰。让并行变快而串行变慢的日子还要等到什么时候呢？

还有，处理单元与内存之间的速度差异也在不断地增加。尽管现在的存储密度正以可预见的等同于常规集成度增加的速度持续增加，但其访问速度却因各种物理的、技术的以及市场相关的原因[22]，渐近地落后于计算速度。目前尚不清楚要如何才能显著地减小速度上的差距，因此它只会不断加大。访问内存里的一个字的速度与处理器的处理速度之间可能有上百个周期的差距。仅仅在几年前，你才可以购买到在一个时钟周期内即可完成访问的“零等待状态”内存芯片。

内存架构在密度、价格和速度之间出现的不同权衡，已导致了一个日益复杂的内存层次结构。访问一个内存字已演变成一种涉及询问多个缓存层的侦测探查，其中包括从芯片上宝贵的静态 RAM（随机存取存储器）开始，然后可能一直延续到大容量存储器。反之，一个给定的数

自在读书 www.zizidiary.com

据可能会重复地存在于整个缓存层次结构的多个地方，这进而对编程模型也产生了影响。我们不能再将内存当成是一个大的、单一的、可被系统里的所有处理器轻松共享的数据块：缓存促进了局部内存传输，并且让共享数据给人以一种越来越难以维护的错觉[37]。

根据最新的有关新闻报道，光的速度已确定为保持在恒定的（可以认为它就是不可变的）大约 300 000 000 m/s。在氧化硅里光的速度（与当今芯片内部的信号传播相关）大约只有一半，并且今天我们可以达到的实际数据传输速度远低于理论极限值。这意味着在高频段全局互连会有更多的麻烦。如果我们想要构建一个 10 GHz 的芯片，那么在理想情况下，仅仅是将一个位传输通过一个 4.5 cm 宽的芯片也需要花费 3 个周期，同时还未实际执行任何计算。

总之，我们正朝着高密度和巨大计算能力相融合的处理器发展方向前进。然而，它们正变得越来越孤立和难以实现与使用，因为要受限于互连性、信号传播速度和内存访问速度。

计算机行业围绕这些障碍正顺其自然地发展着。现在有这样一种迹象，即在给定的计算能力下，要求在尺寸和耗能方面进行压缩。现今的便携式数码辅助设备很是让人沉迷，仅仅使用 5 年前的技术不可能组装出与之相同尺寸和功能的设备。不过，如今的趋势对传统计算机来说帮助不大，尤其是它们还想在保持大致相同尺寸下实现提升计算能力的目的。基于这些原因，芯片制造商决定放弃实现更快时钟频率，取而代之的是决定提供以已知的方式包裹后的计算能力：多个相同的中央处理单元（CPU）通过总线彼此连接在一起和连接到内存。这样，在短短的几年时间里，让电脑变得更快的责任很大程度上便从硬件人群转移到了软件人群。使用更多的 CPU 看起来是个不错的提议，但对于普通台式机，如果强行加入大约 8 个以上的处理器，其工作负荷便会变得很脆弱。未来的趋势规划是按指数级的形式对可用的 CPU 数量进行扩张，一直到数十、数百、数千个。为了让给定的程序提速，还需要很多辛苦的编程工作更好地利用这些 CPU。

计算机行业一直都在经历着因各种技术和人为因素所引发的变动和摇摆，但这次我们似乎到了这根绳索的尽头。因为就在不久前，你可能还在为提高程序的速度而没法选择放假休息。这是一桩丑闻，是一种暴行，是并发门（concurrentgate）。

13.2　数据共享简史

有一个发生在计算领域的转变，即突然间处理和并发模型都在发生着变化，特别是与编程语言和范型的发展速度相比更是明显。编程语言及其相关的样式花了数年乃至数十年的时间才成为社区不可磨灭的传说，而在并发性方面发生的变化最早也只能追溯到 21 世纪初。

例如，我们不久以前对普通并发性[①]的理解仍然围绕着分时（time sharing）操作，而它可以依次追溯到 20 世纪 60 年代的大型机。在当时，CPU 的时间非常宝贵，为提高整体利用率，让源自多个控制台的多个程序共享 CPU 就变得很有意义。进程（process）指的是一个正运行着的程序所拥有的状态和资源。为实现分时，CPU 需要使用一个定时器中断连同一个软件调度器一起工作。在每一个定时器中断上，调度器决定哪个进程应该获得下一轮的 CPU 时间，这样便产

① 后面讨论的重点是普通并发性，并不会讨论向量操作并行化和其他专门的并行内核。

自在读书 www.zizidiary.com

生了多个进程同时在运行的假象，而事实上它们都在使用同一个 CPU。

为了防止充满错漏的进程干扰到另一个进程以及操作系统代码，人们引入了一项被称为“硬件内存保护”（hardware memory protection）的技术。在当今的系统里，内存保护与“内存虚拟化”（memory virtualization）被合并在一起，以确保健壮的进程隔离：每个进程都认为自己“拥有”机器内存，而实际上是有一个从逻辑地址（进程所看到的内存）到物理地址（机器访问的内存）的转换层负责协调所有进程与内存的交互，并将进程彼此隔离开来。这种做法的好处在于，失控的进程只会对自己造成破坏，而对其他进程或操作系统内核不会产生影响。坏处在于，每次的任务切换都一定会产生一个代价高昂的地址转换工具的交换；此外，刚切换过来的进程在唤醒时会出现缓存失效的问题，因为全局共享缓存最可能被其他进程使用。线程（thread）因此而诞生。

线程是一个不带相关地址转换信息（一种赤裸的执行环境：处理器状态加上栈）的进程。多个线程共享一个进程的地址空间，这意味着线程在启动和切换时代价会相对低一些，并且它们彼此很容易和廉价地实现数据共享。在一个 CPU 上进行跨线程内存共享可能是最简单的——一个线程写入，另一个读取。利用分时，数据被一个线程写入的次序自然跟其他线程所看到写入顺序是一致的。对更高级别数据不变量的维护，可通过互锁机制来保障，例如，通过同步原语（如信号量和互斥）可实现对临界区的保护。在 20 世纪后期，围绕着什么才可被称为“经典的”多线程编程，出现了大量的看法、民间说法和趣闻。其主要的特点是共享地址空间、简单的记忆效应可见性规则，以及互斥驱动的同步。当然也存在其他的并发模型，但经典多线程在主流硬件上最常用。

今天的主流命令式语言，如 C、C++、Java 或 C#，都是在经典多线程的年代开发的——那是个美好的旧时光：简单的内存架构、简单的数据共享，以及易于理解的互锁原语。很自然，这些语言造就了这样一个现状，即硬件会负责协调所有共享同一内存的线程。要知道，正是那个多线程定义强制要求所有线程都共享同一个地址空间，这一点与操作系统进程不同。此外，消息传递 API（如 MPI 规范[29]）也已经有了库的形式，而它最初用于如（超级）计算机集群那样的高端硬件。

在同一历史时期，后来才出现的函数式语言则采取了一种基于数学纯度的原则性立场：我们对基于硬件的建模不感兴趣，他们是这样说的，我们要的是基于数学的建模。数学的大部分是不变的，且具有时间不变性，这让它成为理想的并行化的候选对象。想象一下，当第一代从数学家转变过来的程序员听说并发性时会怎么样呢？他们一定会拍打着自己的额头说：“等一下！”。特别需要指出的是，在函数式编程圈子里，这种计算模型天生就支持无序、并发的执行。但这种潜能更多只是一种潜在的资源，直到近期才成为实现的目标。

最后，从 20 世纪 80 年代后期开始，Erlang 作为一个用于通信应用程序这样一个特定领域的嵌入式语言被开发出来。该领域需要成千上万的同步程序运行在同一台机器上，并且特别钟情于利用消息传递的、“发后不管”（fire-and-forget）的通信方式。尽管主流硬件和操作系统对这样的工作未进行优化，但 Erlang 最初是运行于专门的硬件上的。其结果就是产生了这么一种语言，它最早将繁重的并发性能力与坚固的基于消息传递的、无共享的通信方式相结合，并形成了一种不纯的函数式风格。

快进到 21 世纪 10 年代。今天，即使普通的机器也有了多个处理器，这 10 年的主要挑战

自在读书 www.zizidiary.com

是将更多的 CPU 插到芯片上。这产生了大量的影响，其中最重要的就是无缝共享内存的消亡。

一个分时 CPU 只附带一个内存子系统——有了多个缓冲区，即多级缓存，这里有它的工作。不管 CPU 怎么分时，读取和写入都是通过同一个管道进行的。这样，便能跨所有线程对内存统一视图进行维护。相比之下，多个相互连接的 CPU 无法共享这个缓存子系统：这种缓存需要进行多通道存取（代价非常高且可伸缩性差），而且很难同时放置在所有 CPU 的附近。因此，现今的 CPU 全部都带有自己的专用高速缓存。将 CPU 与缓存这对组合连接在一起的硬件和协议都是影响多处理器系统性能的关键因素。

多个缓存的存在让跨线程数据共享变得更加困难。现在不同线程里的读取和写入可能会发生在不同的缓存里，因此从一个线程到另一个线程的共享数据不再那么直接。事实上，它变成了一种有序的消息传递[①]：对所有这种共享，在缓存子系统中都必须进行握手操作，这样才能确保源自最新写入者的共享数据正确地传输到读取者和主内存。

好像事情变得没那么有什么意思了，缓存同步协议有了新的情况：它们操纵数据的方式是按块进行的，而不是单独地按字读取和按字写入。这意味着通信处理器会“忘记”数据写入的确切顺序，进而会导致矛盾行为，即明显会颠倒因果关系和已有常识：一个线程先写入 x，然后写入 y；一会儿之后，另一个线程会看到那个新的 y，但看到的 x 是旧的。这种有违因果关系的行为很难集成到经典多线程的一般模型里，在经典多线程里充斥着的是对时间片的直觉和简单的内存模型。即使在经典多线程方面最厉害的程序员，也会发现很难调整他们的编程风格和模式以适应新的内存架构。

为了说明在当今的并发性世界里快速发生的变化，以及数据共享对语言的并发性方法产生的严重影响，请仔细阅读下面这段建议，它是优秀书籍 *Effective Java* [8，204 页第 51 项]的 2001 年版里给出的：

> 当多个线程都可运行时，线程调度器会决定哪个线程应该运行以及它要持续运行多久……想要编写出健壮的、快速响应的、便携式的多线程应用程序，可采用的最好方式是确保在任何给定的时间可运行的线程都是很少量的。

对于现今的观察者，有一个细节让人有点吃惊，即单处理器的、分时间片的线程不仅被前面的这段引用提及，而且还在未说明的情况下进行了假定。于是，这本书的 2008 版[②][9]将这个建议更改为“确保可运行线程的平均数量不要显著大于处理器的数量”。有意思的是，就算是这个建议，它看起来很合理，但还是使用了好些未加说明的假设：第一个假设是，在线程之间存在很激烈的数据竞争，进而因互锁开销导致性能下降；第二个假设是，在可以执行这个程序的机器上，处理器的数量大不相同。鉴于此，在书籍 *Programming Erlang* [5，第 20 章]里，用最强烈的措辞反复地给出了与之相反的建议：

> 大量使用进程　这一点很重要——我们必须让 CPU 保持繁忙。所有的 CPU 随时都必须保持忙碌。实现这一目标最简单的方法是大量使用进程③。当说到大量进程时，我所要表达的意思是与 CPU 的数目有密切关系。如果有了大量进程，我们就无须担心要

① 这个具有讽刺意味，因为在经典多线程的时候，共享内存已经比消息传递更快了。
② 为反映出线程仅仅是一种并发模型的事实，该主题的标题从“线程”更改为“并发性”。
③ Erlang 进程与 OS 的进程是完全不同的。

自在读书 www.zizidiary.com

保持 CPU 繁忙。

哪个建议是正确的呢？通常情况下，需要视具体情况而定。第一个建议在 2001 年以前的老式硬件上非常有效；第二个建议则在密集的数据共享环境以及由此带来激烈的数据竞争场景中很有效；第三个建议则最适合于低竞争，高 CPU 计算的情形。

因为共享内存变得越来越难，当今的趋势是让数据共享变得更细，以及青睐于函数式消息传递的方法。顺便提一句，近年来对 Erlang 以及其他用于并发应用的函数式语言的兴趣正在不断增加。

13.3　快瞧瞧，没有（默认）共享

近年来随着硬件和软件的发展，D 语言选择的做法与其他命令式语言截然不同：是的，D 语言支持线程，但它们默认不会共享任何可变数据——它们彼此间是隔离的。这种隔离不像进程那样通过硬件实现，也不是通过运行时检查来实现，它是由设计 D 语言的类型系统所采用的方式自然形成的结果。

这种决定受到了函数式语言的启发，函数式语言也曾努力禁止所有更改以及由此产生的可变共享。其中有两个差异：第一个差异是，D 语言程序仍然可以自由地对数据进行更改——只是可变数据对其他线程来说是不可随意访问的；第二个差异是，无共享只是一种默认选择，而不是唯一选择。为了定义跨线程共享的数据，必须要使用 shared 来修饰其类型。例如，在模块作用域有两个简单的定义：

```
int perThread;
shared int perProcess;
```

在大部分语言里，第一个定义（或者与之等效语法）会引入一个可被所有线程使用的全局变量。然而，在 D 语言里，perThread 在每个线程里都有一个单独的副本。第二个声明只会分配一个 int，它共享于所有的线程，因此在某种程度上它更接近（但本质上不同）传统的全局变量。

变量 perThread 会使用名叫线程局部存储（thread local storage，TLS）的操作系统工具进行存储。由 TLS 分配的数据，其访问速度取决于编译器实现和底层的操作系统。与 C 语言程序里的常规全局变量相比，其慢的程度几乎可以忽略不计。在极少的情况才会关心这个问题，例如，在访问频繁的循环里，你可以将这个全局变量加载到一个栈变量里。

这种方案有两个很重要的优势。第一个优势是，默认支持共享的那些编程语言都必须要很仔细地同步访问数据。这一点对于 perThread 来说是没必要的，因为它对每个线程都是私有的。第二个优势是，修饰符 shared 意味着类型系统和人类用户都知道 perProcess 会同时被多个线程访问。尤其是，类型系统会积极地守护 shared 数据的使用情况，并且会禁止那些明显错误的用法。这种方案将传统方案给颠覆了：在默认共享的规则里，程序员必须要手工跟踪哪个数据被共享，而哪个数据又未被共享。事实上，大部分并发相关的错漏都是由于过度和无保护的共享造成的。在显式共享规则里，程序员能够确切地知道未标记为 shared 的数据事实上绝不会出现多个线程看到它的情况。为了确保这个保证，shared 值会进行一些附加的检查，我们很快会谈到这一点。

shared 数据的应用仍然是一个高级话题，因为尽管底层相关内容会由类型系统自动提供

自在读书 www.zizidiary.com

担保，但高层的不变量则可能没有这种担保。为提供安全、简单和高效的线程间通信，首选的方法是使用一种被称为“消息传递”的范型。内存隔开的线程彼此间通过发送异步消息进行交流，这种消息只是简单地由打包在一起的 D 语言值构成。

孤立工作线程通过简单通道进行交流是一种非常健壮的、经时间考验过的并发性方法。多年来，Erlang 都是这样做的。所采用的具体方式是让应用程序都构建在信息传递接口（Message Passing Interface，MPI）规范[29]基础上。

锦上添花[①]的是，即使是在默认支持共享的多线程语言里，好的编程实践实际上都会将“隔离线程”奉为信条。Herb Sutter，并发性方面的一位世界级专家，在一篇富于表现力的文章：“正确使用线程=隔离+异步消息”[54]中写道：

> 线程是一种用于表达异步工作的底层工具。需要通过准则“提升”它们的级别：尽最大努力让它们的数据成为私有的，并且使用异步消息进行同步和交流。每一个需要从其他线程或从人们那里获得信息的线程都应该有一个消息队列（可以是简单的 FIFO 队列也可以是具有优先级的队列），并围绕着一条事件驱动的消息主管线开展工作。使用事件驱动逻辑来代替混乱不堪的处理方法对提高代码的清晰性和确定性是一种非常好的方法。

如果说这几十年的计算给了我们什么样的教训，那一定是面向准则（discipline-oriented）的编程还未形成规模。然而令人欣慰的是，前面的引用相当准确地总结了后面几节的内容，也就只剩下准则部分未包括。

13.4 启动线程

为启动一个线程，请使用 spawn 函数：

```
import std.concurrency, std.stdio;

void main() {
    auto low = 0, high = 100;
    spawn(&fun, low, high);
    foreach (i; low .. high) {
        writeln("Main thread: ", i);
    }
}

void fun(int low, int high) {
    foreach (i; low .. high) {
        writeln("Secondary thread: ", i);
    }
}
```

spawn 函数接受一个函数地址&fun，以及多个参数$\langle a_1\rangle$, $\langle a_2\rangle$, ..., $\langle a_n\rangle$。参数数量 n 和它们的类型都必须与函数 fun 的签名相匹配，即调用 fun($\langle a_1\rangle$, $\langle a_2\rangle$, ..., $\langle a_n\rangle$)必须是正确的。在编译时会执行这个检查。spawn 会创建一个新的执行线程，此线程发起调用 fun($\langle a_1\rangle$, $\langle a_2\rangle$, ..., $\langle a_n\rangle$)，

① 原文为“To add acclaim to remedy”，它是短语“to add insult to injury”（雪上加霜）的反义词，故译作“锦上添花”。——译者注

自在读书 www.zizidiary.com

然后终止。当然，spawn 不等待线程终止——在线程被创建，以及传递参数给它（在本例里为两个整数）之后，它会立即返回。

上面的程序会将共计 200 行的内容输出到标准输出设备。行的交错情况取决于多种因素：有可能看到的是来自主线程的 100 行，后面紧跟着来自辅线程的 100 行；或者是与之恰好相反的情况；或者为看似随机的交错。不过，绝不存在两条消息混合在同一行的情况。这是因为在定义 writeln 时，已保证对输出流的每一个调用都是原子的。还有，对于每个线程输出的那些行，它们的顺序都会被保持。

即使 main 函数的执行可能在处于辅线程里的 fun 函数执行之前结束，此程序在退出之前也会耐心地等待所有线程执行完成。这是因为运行时支持库遵循了一个很小的程序终止协议，我们稍后会讨论到这一点。现在只是提醒注意一下，其他线程并没有因为 main 函数的返回而突然死掉。

正如隔离保证所承诺的，新创建的线程与调用者之间并没有共享任何内容。几乎是什么都没共享：全局的文件句柄 stdout 事实上被两个线程共享。也没什么好隐瞒的：如果检查一下 std.stdio 模块的实现，你就会看到 stdout 被定义成一个全局共享变量。类型系统里的所有情况都得到了充分的考虑。

immutable 共享

通过 spawn 可以调用什么样的函数呢？坚持无共享的立场强加了某些限制——只能将按值传的参数用于线程启动函数（即上面示例中的 fun）。所有按引用的传递，不管是显式的（使用 ref 参数）还是隐式的（如使用数组）都是被禁止的。鉴于此，我们来看看该示例重写后的样子：

```
import std.concurrency, std.stdio, std.exception;

void main() {
    auto low = 0, high = 100;
    auto message = "Yeah, hi #";
    spawn(&fun, message, low, high);
    foreach (i; low .. high) {
        writeln("Main thread: ", message, i);
    }
}

void fun(string text, int low, int high) {
    foreach (i; low .. high) {
        writeln("Secondary thread: ", text, i);
    }
}
```

重写后的示例与原来很相似，但它会输出一个附加字符串。这个字符串是在主线程里创建的，并且在传递时并没有将其复制到辅线程。实际上，message 的内容在两个线程间是共享的。这违反了上述原则，即所有数据共享都必须通过使用 shared 关键字显式地进行标记。然而，这个示例却能够编译和运行。到底是什么情况？

在第 8 章里解释过，即 immutable 会提供很强的保证：immutable 值保证在其整个生命周期里绝不会被改变。在第 8 章里还对此进行了解释（参见 8.2 节）：string 类型实际上是

自在读书 www.zizidiary.com

immutable(char)[]的别名。最后，我们知道所有的竞争都是由共享可写数据引起的——只要没人更改它，你就可以随意地共享数据，因为大家看到的内容都是相同的。类型系统和整个线程基础设施都认可这一事实，它们允许所有 immutable 类型数据都可以自由地跨线程共享。特别地，string 类型可以共享，因为它们的字符不会被更改。事实上，之所以将 immutable 引入 D 语言，其背后的主要动机就是想让它有助于实现结构化数据的跨线程共享。

13.5　线程间的信息交换

随意交错输出消息的线程几乎是没有意义的。我们来修改一下这个示例，让它们协同工作，一起输出类似下面的信息：

```
Main thread: 0
Secondary thread: 0
Main thread: 1
Secondary thread: 1
...
Main thread: 99
Secondary thread: 99
```

为了实现这一目标，需要在这两个线程之间定义一个小协议：主线程应该将“输出这个数字”这样的消息发送给辅线程，而辅线程必须回答“完成打印”。这里几乎没什么并发操作，但此示例能很好地达到解释纯通信的目的。在实际应用程序里，线程大部分时间应花费在完成有用的工作上，而彼此间的通信开销则相对要少很多。

首先，为了让两个线程进行通信，它们需要知道如何去称呼对方。一个程序可能有多个线程在对话，因此对其进行标识是很有必要的。为了与某个线程通信，必须要获得它的“线程 id”，也被称做“tid”，它由 spawn 返回（tid 的类型名字实际为 Tid）。相应地，辅线程也需要一个 tid 以将响应发送回去。这一点比较容易实现，方法是让发送方指定自己的 tid，它跟在“蜗牛式”的邮件信封上写下发件人地址的方式一样。代码类似下面的样子：

```
import std.concurrency, std.stdio, std.exception;

void main() {
    auto low = 0, high = 100;
    auto tid = spawn(&writer);
    foreach (i; low .. high) {
        writeln("Main thread: ", i);
        tid.send(thisTid, i);
        enforce(receiveOnly!Tid() == tid);
    }
}

void writer() {
    for (;;) {
        auto msg = receiveOnly!(Tid, int)();
        writeln("Secondary thread: ", msg[1]);
        msg[0].send(thisTid);
    }
}
```

这次 writer 不再接受任何参数，因为它是以消息的形式接收所需的信息的。主线程会保

自在读书 www.zizidiary.com

存被 spawn 返回的 Tid，然后在对方法 send 的调用里使用它。这个调用会将两个数据发送到其他线程：当前线程的 Tid（它可通过全局特性 thisTid 进行访问），以及要输出的那个整数。在将数据抛到其他线程之后，主线程会等待确认，而等待的形式就是调用 receiveOnly。函数 send 和 receiveOnly 会协同工作：在某个线程里的 send 调用会与其他线程里的 receiveOnly 调用相遇。receiveOnly 这个名字里包含“only”是有道理的，这是因为 receiveOnly 只会接受特定类型。例如，在调用 receiveOnly!bool()之后，调用者只会接受由一个 bool 值构成的消息。如果另一个线程又发送了其他的内容，那么 receiveOnly 便会抛出一个 MessageMismatch 异常。

我们暂时别管正在翻找 foreach 循环的那个 main 函数，集中精力转去看看那个 writer 函数的实现，它实现了那个迷你协议的另一半。writer 将时间都花在了一个循环里。开始时，它会收到一条一定由一个 Tid 和一个 int 所构成的消息，而这正是 receiveOnly!(Tid, int)()调用所保证的。同样地，如果主线程发送一条消息，而该消息带有不同数目或类型的参数，那么 receiveOnly 会失败，并抛出一个异常。如前所述，writer 的 receiveOnly 调用要与 main 的 tid.send(thisTid, i)调用完全匹配。

msg 的类型为 Tuple!(Tid, int)。通常情况下，带有多个参数的消息会被封装在 Tuple 里，每一个参数都对应一个成员。然而，如果这个消息只包含一个值，那么封装在 Tuple 里就会显得多余。例如，receiveOnly!int()返回的是 int，而不是 Tuple!int。

继续来看 writer，接下来的一行会执行实际的输出。回想一下，对于元组 msg，msg[0]会访问第一个成员（即 Tid），而 msg[1]会访问第二个成员（即 int）。最后，为了确认它已完成了控制台的输出，writer 会简单地将自己的 Tid 发送回 sender——类似于一封仅仅是为证实发信地址的空白信件。这封空的信件是这样进行暗示的，“是的，我收到了你的消息，并且采取了相应的动作。现在轮到你了。”主线程在继续其工作之前会一直等待这个确认，然后循环继续往下执行。

在这里，将辅线程的 Tid 发送回去显得有些多余，只要一个任意的假值（如 int 或 bool）即可。但在一般情况下，会有许多线程彼此发送消息，因此自我标识变得特别重要。

13.6　receive 的模式匹配

大部分有用的通信协议都比我们上面定义的协议复杂，并且 receiveOnly 的功能相当有限。例如，使用 receiveOnly 很难实现像“接收整数或字符串”这样的动作。

有一个功能更强大的原语，即 receive，它可以根据消息的类型对它们进行匹配和派发。典型的 receive 调用像这个样子：

```
receive(
   (string s) { writeln("Got a string with value ", s); },
   (int x) { writeln("Got an int with value ", x); }
);
```

上面的调用可以匹配下面所有的 send 调用：

```
send(tid, "hello");
send(tid, 5);
```

自在读书 www.zizidiary.com

```
send(tid, 'a');
send(tid, 42u);
```

第一个 send 调用会匹配到一个字符串，因此它会被派发到 receive 里的第一个函数字面量。而另外三个调用会匹配到一个整数，并且会被传递给第二个函数字面量。顺便说一下，那些处理器函数并不要求是字面量——也可以为命名函数的地址：

```
void handleString(string s) { ... }
receive(
   &handleString,
   (int x) { writeln("Got an int with value ", x); }
);
```

匹配不一定是精确的。实际上，它遵循的是常规的重载规则，即 char 和 uint 会隐式转换为 int。与此相反的是，下面的调用不会被匹配到：

```
send(tid, "hello"w);    // UTF-16 字符串（参见 4.5 节）
send(tid, 5L);          // long
send(tid, 42.0);        // double
```

当 receive 看到一条未知类型的消息时，它并不会抛出异常（像 receiveOnly 一样）。消息传递子系统只会单纯地将无匹配的消息保存在一个队列里，俗称线程的“邮箱”（mailbox）。receive 会耐心地等待与邮箱里消息的类型相匹配的消息到来。这让 receive 和基于它的协议实现起来更为灵活，但也更容易造成阻塞和让邮箱拥挤不堪。只要有一个通信存在误解，便会让线程的邮箱累积好些错误类型的消息，而这时 receive 却还在等待一个绝不会到达的消息类型。

将 Tuple 作为一种中间工具，send/receive 这一对组合可以轻松地处理多个参数。例如：

```
receive(
   (long x, double y) { ... },
   (int x) { ... }
);
```

等同于下面的内容：

```
receive(
   (Tuple!(long, double) tp) { ... },
   (int x) { ... }
);
```

调用 send(tid, 5, 6.3)会匹配上面两个示例里的第一个函数字面量。

为了让线程在消息被延迟时可以采取一些应急动作，receive 还存在一种变体，即 receiveTimeout，它可以在指定时间之后终止。在终止时，receiveTimeout 会通过返回 false 进行提醒：

```
auto gotMessage = receiveTimeout(
   1000, // 时间单位毫秒
   (string s) { writeln("Got a string with value ", s); },
   (int x) { writeln("Got an int with value ", x); }
);
if (!gotMessage) {
    stderr.writeln("Timed out after one second.");
}
```

自在读书 www.zizidiary.com

13.6.1　首次匹配

假设有下面这样一个示例：

```
receive(
   (long x) { ... },
   (string x) { ... },
   (int x) { ... }
);
```

这个调用无法编译：receive 会拒绝这个调用，因为第三个处理器绝不可能到达。所有 int 都会被发送到第一个处理器所在的那个管道。

在 receive 里，参数的顺序决定了匹配是如何进行尝试的。例如，它有点像 try 语句里 catch 子句所进行的匹配，但与面向对象的函数派发有所不同。理性的人们对首次匹配和最佳匹配的相关品质可能有些不太认同。我只想说，首次匹配能够很好地满足 receive 的这种特殊形式的需要。

receive 执行的编译时强制要求很简单：在 receive 的调用里，对于所有消息类型‹Msg_1›和‹Msg_2›，如果‹Msg_2›的处理器在‹Msg_1›之后发挥作用，那么 receive 会确保‹Msg_2›不可转换到‹Msg_1›；如果能够转换，则意味着‹Msg_1›会匹配到‹Msg_2›类型的消息，从而导致此调用会被拒绝编译。在上面的示例里，当‹Msg_1›为 long，而‹Msg_2›为 int 时，该检查会失败。

13.6.2　匹配所有消息

如果想要保证能查看到邮箱里所有类型的消息，例如，为保证它不会被垃圾邮件填满，该怎么办呢？

答案很简单——只需要在 receive 的最后位置接受 Variant 类型即可，例如：

```
receive(
   (long x) { ... },
   (string x) { ... },
   (double x, double y) { ... },
   ...
   (Variant any) { ... }
);
```

Variant 类型定义在 std.variant 模块里，它是一种动态类型，能恰好容纳下一个其他任何类型的值。receive 会将 Variant 识别为任何消息类型的通用容器，因此在对拥有 Variant 处理器的 receive 进行调用时，只要其队列里存在消息它便会立即返回。

在消息处理“食物链”的底部植入一个 Variant 处理器，是一种非常好的做法。它可以用来确保迷失的消息不会残留在你的邮箱里。

13.7　文件复制——稍有改变

我们来编写一个用于文件复制的小程序——这是一种流行的用于了解一门语言的文件系统接口的方式。K&R 的经典实例 getchar/putchar[34，第 1 章，第 15 页]很讨人喜欢。当

自在读书 www.zizidiary.com

然，系统提供的用于复制文件的程序使用了带缓冲的读取和写入，以及其他许多优化方法来加快传递速度。因此，很难编写出一个可以与之抗衡的程序，但是现在并发性带来了希望。

通常的文件复制方法像下面这个样子：

（1）将数据从源文件读取到一个缓冲区；

（2）如果什么都没读入，则完成；

（3）将缓冲区内容写入到目标文件；

（4）重复步骤 1。

在添加适当的错误处理之后便完成了一个有用的（如果不要求原创）程序。如果你选择了一个足够大的缓冲区，并且源文件和目标文件都处于同一个磁盘里，那么这个算法的性能近似最优。

现在，各式各样的物理设备都可被当做文件存储器，如硬盘、U 盘、光盘、连接的智能手机和远程连接的网络服务。这些设备有着各不相同的延迟和速度特性，并通过不同的硬件和软件接口连接到计算机。这些接口可以并且也应该能够并行地工作，而不是像上面提供的“读取缓冲区/写入缓冲区”算法那样每次只能执行一个动作。在理想的情况下，源设备和目标设备都应该尽可能保持忙碌。于是，我们可以利用遵从“生产者-消费者”协议的两个线程来达到这个效果：

（1）产生一个辅线程，将它用于监听包含内存缓冲区的消息，并将这些信息在一个循环里写入到目标文件；

（2）在新分配的缓冲区里，读取来自源文件的数据；

（3）如果什么都没读入，则完成；

（4）发送一个包含读取缓冲区的消息到辅线程；

（5）重复步骤 2。

在这个新方案里，一个线程可以让源设备保持忙碌，而另一个线程则可以让目标设备保持忙碌。尽管源设备和目标设备的特点各不相同，但速度都可以获得显著提高。如果设备速度是可比较的，并且与内存总线的带宽相比要相对慢些，那么复制的速度理论上可以翻倍。我们来编写一个简单的“生产者-消费者”程序，它实现的是将 stdin 复制到 stdout：

```
import std.concurrency, std.stdio;

void main() {
    enum bufferSize = 1024 * 100;
    auto tid = spawn(&fileWriter);
    // 读取循环
    foreach (immutable (ubyte) [])buffer; stdin.byChunk(bufferSize)) {
        send(tid, buffer.idup);
    }
}

void fileWriter() {
    // 写入循环
    for (;;) {
        auto buffer = receiveOnly!(immutable(ubyte)[])();
        stdout.rawWrite(buffer);
    }
}
```

上面的程序会通过 immutable 共享将主线程的数据传送到辅线程：被传递消息的类型为

自在读书 www.zizidiary.com

immutable(ubyte)[]，即由不可变无符号字节构成的数组。这些缓冲区是在 foreach 循环里通过读取类型为 immutable(ubyte)[]的输入块而获得的，每一个的大小为 bufferSize。每循环一次时，都会执行分配新缓冲区、读入数据并将其绑定到 buffer 这样一些操作。foreach 的控制部分做了大量的艰苦工作。循环体所要做的是将缓冲区发送到辅线程。如前所述，随处传递是有可能的，因为它的类型为 immutable。如果将 immutable(ubyte)[]替换成 ubyte[]，则 send 调用将无法编译。

13.8　线程终止

到目前为止，所给的示例都还存在好些不同寻常的地方，特别是在 13.5 节定义的 writer 和在 13.7 节定义的 fileWriter：这两个函数都包含了一个无限循环。实际上，仔细看看这个文件复制示例便会发现：在复制文件这件事情上，main 和 fileWriter 彼此都是非常了解的，但它们从未谈及应用程序终止的事。换句话说，main 还未告诉过 fileWriter："我们干完了！收拾一下回家去吧！"

多线程应用程序的终止一直都是很复杂的。线程的启动比较容易，但一旦它们启动就很难结束。应用程序的关闭事件是异步的，并且可能在一个很随意的操作过程中捕捉到线程。底层的线程 API 提供了一种方式，可用于强制终止线程，但请注意：这个函数并非"利器"，应该替换成更高级的关闭协议。

D 语言提供了一个简单而又健壮的线程终止协议。每个线程都拥有一个所有者线程。默认情况下，这个所有者就是初始化 spawn 的那个线程。可以调用 setOwner(tid)动态更改当前线程的所有者。每个线程都恰好拥有一个所有者，但给定的线程可以拥有多个线程。

所属关系最重要的体现在于：当所有者线程终止时，其内部的 receive 调用会抛出 OwnerTerminated 异常。只有在 receive 没有再匹配到信息，并且一定是在等待某个新消息时，才会抛出此异常；只要 receive 还能从邮箱里获取到某些信息，就不会抛出它。换句话说，当所有者线程终止时，对于所有从属线程的 receive 调用（或者是 receiveOnly 调用也一样），只要它们会出现阻塞等待新消息的情况，就会导致 OwnerTerminated 异常抛出。所属关系不一定是单向的。事实上，两个线程甚至可能相互拥有。在这种情况下，无论哪个线程完成，都会通知另一方。

有了线程所属关系之后，我们再用全新的眼光来看看 13.7 节的文件复制程序。在任何给定的时刻，在主线程和辅线程之间总是有许多消息正在传送。相对于写入速度来说，读取速度越快，在写入者线程的正等待处理的邮箱里等特的缓冲区便会越多。当 main 返回时，这将导致 receive 调用抛出异常，但在所有等待消息被处理之前异常不会被抛出。就在写入者的邮箱被清除之后（并且最后一块数据被写入到目标文件），接下来的 receive 调用会抛出异常。写入者线程随着 OwnerTerminated 一起退出，而运行时系统会识别出这个异常，并简单地忽略它。操作系统像往常一样关闭 stdin 和 stdout，于是复制操作顺利完成。

最后一条消息从 main 发送出去的时刻与 main 返回（导致 receive 抛出异常）的时刻，这

自在读书 www.zizidiary.com

两者之间似乎存在某种竞争。如果此异常“发生”在最后那条消息之前（或者更糟的情形，如发生在最后几条消息之前），那么结果会怎么样呢？事实上，并不会出现竞争，因为在发起线程里，人们总是会遵从因果关系：最后一条消息会在抛出 OwnerTerminated 异常之前被发送到辅线程队列里（事实上，这个异常正是被当做常规的消息通过同一个队列传播的）。不过，在 main 退出时，如果还有另外一个第三方线程正在往 fileWriter 的队列发送消息，那么竞争就会存在。

类似的推理可以表明，我们之前那个以同步方式输出 200 条消息的简单示例也是正确的：main 会在向辅线程发送完（恰好）最后一条消息之后退出；辅线程首先处理完这个队列，然后在结束时抛出 OwnerTerminated 异常。

如果发现将异常抛出作为处理线程退出的机制不太优雅，也可以显式地处理 OwnerTerminated 异常：

```
// 结束时没有异常
void fileWriter() {
    // 写入 loop
    for (bool running = true; running; ) {
        receive(
            (immutable(ubyte)[] buffer) { tgt.write(buffer); },
            (OwnerTerminated) { running = false; }
        );
    }
    stderr.writeln("Normally terminated.");
}
```

在这种情况下，当 main 退出时，fileWriter 会平静地返回，这样大家都高兴。但是，当辅线程（即写入者）抛出异常时，会发生什么事情呢？如果将数据写入到 tgt 时出现了问题，那么对 write 函数的调用可能会失败。在这种情况下，从主线程调用 send 将会失败，同时会抛出 OwnedFailed 异常，这些正是应该发生的事情。顺便说一下，如果某个从属线程正常退出（相对于异常抛出），那么随后针对该线程的 send 调用也会失败，只是抛出的异常类型有所不同，为 OwnedTerminated。

这个文件复制程序比其简单化版本表现得更为健壮。不过需要说明的是，在线程之间的关系比较简单和容易理解时，依赖于默认终止协议是可以顺畅工作的。当涉及更多线程，并且所属关系图很复杂时，最好还是建立在显式的“结束通信”协议上。在这个文件复制的示例里，还有一个比较简单的想法，即按惯例发送一个 0 大小的缓冲区，以告知写入线程：读取线程已经成功结束。然后，写入线程确认终止读取线程，最终读取线程便可以退出了。这种显式协议对于很多情况（如在读取线程和写入线程之间，存在多个线程处理数据流）都具有很好的可伸缩性。

13.9　带外通信

假设你正在使用我们刚才定义的那个可能有些智能的文件复制程序，将一个大文件从一个快速的本地存储器复制到另一个缓慢的网络驱动器。在复制的中途，突然出现了一个读取错误——文件损坏了。这会导致 read 和随后的 main 抛出异常，而这时有许多正在传输中的缓冲区还未被写入。更一般地，如果所有者线程正常终止，那么所有阻塞的来自附属线程的

自在读书 www.zizidiary.com

receive 调用都会抛出异常。如果所有者线程因为异常而退出，那么会出现什么情况呢？

如果某个线程以异常方式终止，这表明出现了严重的问题，必须以相对紧迫的方式将这一情况告知所有者线程。事实上，这是通过带外消息（out-of-band message）完成的。

先回想一下，receive 只关心匹配的消息，而其他所有的消息则堆积在队列中。这里有一个修正此行为的方法。线程通过调用 prioritySend 而不是 send 来初始化带外消息。这两个函数接受的参数相同，但它们在接收端所表现出的行为却大不相同。使用 prioritySend 传递一个类型为 T 的消息会引发接收线程里的 receive 采取下面的动作。

- 如果 receive 调用能够处理类型 T，那么优先级高的消息将是下一个被处理消息，即使它比其他常规（非优先的）消息晚到达也一样。优先级高的消息总是会被推到队列的开始处，因此最后一个优先级高的发送消息总会在第一个被 receive 获取（即使有其他优先级消息已在等待也一样）。
- 如果 receive 调用不能处理类型 T（即会让此消息在信箱里进行等待），并且如果 T 继承了 Exception，那么 receive 会直接抛出此消息。
- 如果 receive 调用不能处理类型 T，并且 T 未继承 Exception，那么 receive 会抛出一个类型为 PriorityMessageException!T 的异常。这个异常会以一个称为 message 的成员的形式包含被发送消息的副本。

如果某个线程通过异常退出，那么此异常 OwnerFailed 会通过 prioritySend 传播到它所拥有的所有线程。在那个文件复制程序里，main 抛出异常也会引起 fileWriter 在调用 receive 时抛出异常；而整个进程在终止时会输出错误消息并返回一个非零的退出代码。与正常的终止情况不同，在传输过程中可能存在缓冲区已被读取但仍未写入的情况。

13.10　邮箱堵塞

那个“生产者-消费者”式的文件复制程序工作得很好，但也存在一个严重的缺陷。假如要在速度不同的两个设备之间复制一个大文件，将一部合法获得的电影文件从一个内部驱动器（快速的）复制到网络驱动器（可能相当慢）。在这种情况下，生产者（main 线程）会以相当快的速度产生出大量缓冲区，它比消费者将它们输出到目标文件的速度快很多。这两个速度之间的差异会导致缓冲区的“净积累”，这可能会使程序消耗掉很多内存，未能实现效率的提高。

为避免出现邮箱堵塞，并发性 API 允许设置线程消息队列尺寸的最大值，并且还可设置在已达到尺寸最大值时可采取的动作。这里是相关的函数签名：

```
// 在 std.concurrency 内部
void setMaxMailboxSize(Tid tid, size_t messages,
    bool function(Tid) onCrowdingDoThis);
```

只要有新消息传递进来，如果队列里已经包含了 messages 个条目，那么调用 setMaxMailboxSize(tid, messages, onCrowdingDoThis) 便会将并发性 API 导向调用 onCrowdingDoThis(tid)。如果 onCrowdingDoThis(tid) 返回的是 false 或者抛出了

自在读书 www.zizidiary.com

一个异常，那么这条新消息会被忽略；否则，这个线程的队列大小会被再次检查：如果小于 messages，那么新消息会被发送到线程 tid，否则，恢复整个循环。

这个调用是发生在调用者线程里，而不是在被调用线程里的。换句话说，初始发送消息的线程还会负责采取应急行动，以应对收件人邮箱已达到最大尺寸的情况。询问这个调用为什么不是发生在被调用线程里，看起来是很合理的。但是，在具有大量线程的程序里，这是一种错误的实现可伸缩性的方式，因为满邮箱的线程可能会由于有大量来自其他试图发送消息的线程的调用而变得无法正常工作。

当邮箱充满时，可以采取一些预设的动作：阻塞调用者，直到队列变小；抛出异常或忽略新消息。这些预定义的行为被封装成下面这个样子：

```
// 在 std.concurrency 内部
enum OnCrowding { block, throwException, ignore }
void setMaxMailboxSize(Tid tid, size_t messages, OnCrowding doThis);
```

在我们的示例里，一旦邮箱变得太大，最好是简单地阻塞读取线程。在调用 spawn 之后通过插入下面的调用即可达到这样的效果：

```
setMaxMailboxSize(tid, 1024, OnCrowding.block);
```

后面几节描述的是线程间的通信方法，这些方法可用于替代消息传递或作为补充。消息传递是推荐使用的线程间通信方法，它容易理解、快速、表现好、可靠且可伸缩强。只会在一些特殊的情形里，你才会触及底层的通信机制——请记住，“特殊的”并不总是像看上去那样特殊。

13.11 shared 类型修饰符

我们在 13.3 节已经接触过 shared。对类型系统来说，shared 表明的是多个线程可以访问同一块数据。编译器认可这一事实，同时会对共享数据的操作进行限制，并且为这些被接受的操作生成特殊的代码。

```
shared uint threadsCount;
```

上面这个全局定义会引入一个类型为 shared(uint) 的值，它对应于 C 语言程序里的全局无符号整数。这种变量对系统内的所有线程都是可见的。这种说明能为编译器提供很大的帮助：语言“知道”可以从多个线程自由地访问 threadsCount，于是会阻止对它进行单纯的访问。例如：

```
void bumpThreadsCount() {
    ++threadsCount; // 错误!
    // 不能递增共享整数!
}
```

这到底是怎么回事呢？降低到机器那层便可以看到，++threadsCount 不是一个原子操作，它包含了一个“读取-修改-写入”的操作过程：threadsCount 被加载到一个寄存器，该寄存器值递增 1，然后 threadsCount 被写回到内存。整个操作都是正确的，这 3 个步骤需要以不可分割的整体形式执行。正确递增一个 shared 整数的方法是使用专门的由处理器提供的原子递增原语，这些原语都以简便的方式封装在 core.atomic 模块里：

自在读书 www.zizidiary.com

```
import core.atomic;
shared uint threadsCount;

void bumpThreadsCount() {
    // core.atomic定义了
    //     atomicOp(string op)(ref shared uint, int)
    atomicOp!"+="(threadsCount, 1); // 正确
}
```

因为所有的共享数据都会被D语言认真对待，且受到保护，所以允许通过send和receive传递shared数据。

复杂的情况：shared具有传递性

第 8 章解释了 const 和 immutable 为什么必须具有传递性（即可深入的或可递归的）：下面源自 immutable 对象的所有间接量也一定会保持数据为不可变的。否则，immutable 保证在代码里便有了可以进行“圈改”的能力。你不能说某某事物在某种程度上是不可变的，之后它又变成可变的。不过，你可以说这个数据在某种程度上是可变的，在这时它完全可以变成不可变的。深入不变性就像是顺着一条单行道往下走。我们已知道 immutable 推动产生了一些正确的、不会带来麻烦的惯用法，其中包括函数式和跨线程数据共享。如果“在某种程度上”应用了不变性，则能让程序更好地保持正确性。

对于 shared，道理是相同的。事实上，有了 shared 之后，传递性的必要程度就没那么明显了。假设：

```
shared int* pInt;
```

根据修饰符语法（参见 8.2 节），它相当于：

```
shared(int*) pInt;
```

pInt 的正确含义是“指针是共享的，而且此指针指向的数据也是共享的”。有一种浅的、非传递的共享方法可以让 pInt 成为“一个指向非共享内存的共享指针”，如果这种想法可行那就太好了。好比有人这么说：“我要与大家共享这个钱包，只是提醒一点，即里面的钱可不共享。”[①]指针是跨线程共享的，但被指向的数据却不是，这样的声明会把我们带回到美妙的“靠信用制度进行编程”的范型里，而这经过历史证明是彻底失败的。它不是自愿的恶意使用，它是实实在在的错误，它会产生大量的问题。软件是大型的、复杂的、不断变化的，但这些特征从未通过约定顺利地得到担保。

不过，这里还有一个“指向共享数据的非共享指针”的概念，它是合乎逻辑的。有些线程拥有一个私有指针，而这个指针所“盯着的”是共享数据。在语法上，它很容易表达成这个样子：

```
shared(int)* pInt;
```

题外话，如果存在一个“最佳形式服从功能”（Form-Follows-Function）奖的话，那么符号 qualifier(type) 当之无愧。它实在太完美了。即使是在语法上你也无法创建错误的指针类

① 顺便说一句，借助于 const，你可以共享钱包，而里面放置的是具备盗窃保护的钱币，操作方法就是使用类型 shared(const(Money)*)。

自在读书 www.zizidiary.com

型，因为它看起来会像是下面这个样子：

```
int shared(*) pInt;
```

这是无意义的，即使在语法上也一样，因为(*)不是一种类型。

shared 的传递性不仅适用于指针，也适用于结构的字段和类对象：shared 对象的字段都会自动被修饰成 shared 类型。具体细节则会在本章后面一点讨论，而讨论的方式为 shared 如何与类和结构进行交互。

13.12　shared 数据操作及其效果

共享数据的处理有些独特，因为任何时刻都可能会有多个线程对它进行读取和写入。因此，编译器要确保所有操作都能保持数据的完整性，同时还要保证这些操作的因果关系。

支持读取和写入 shared 值，以及能保证这些操作的原子性的类型包括数字类型（real 除外）、指针、数组、函数指针、委托以及类引用。结构类型如果恰好包含了刚提到的这些类型中的某一个，那么它也支持原子式读取和写入。很明显，real 被排除在外，因为它是唯一平台相关的类型。就原子共享问题，其实现已有所考虑。在 Intel 机器上，real 为 80 位，这一点让它在 32 位程序上很难进行原子性的分配。总之，real 主要是用于高精度的临时结果，而很少用于数据交换，因此共享它是没什么意义的。

对于所有数字类型和函数指针，使用 shared 修饰的值与未使用它修饰的值之间可以隐式地来回转换。在 shared(T*)和 shared(T)*之间也可以进行双向指针转换。在 core.atomic 里的原语允许对共享的数字类型进行算术运算。

共享数据的顺序一致性

关于跨线程共享数据操作的可见性，D 语言做了下面两个担保。

- 在某个线程发起的共享数据读取和写入顺序与源代码里指定的顺序相同。
- 共享数据的全局读取和写入顺序是多个线程读取和写入操作的交替顺序。

这似乎是一套很合理的假设——甚至有些不言而喻。事实上，这两个保证只是很符合单处理器系统里实现的时间分片式的线程。

然而，在多处理器系统里，这些担保会变得非常严格。其问题在于，为确保这些担保，所有写操作在全部线程里都一定是立即可见的。为达到这个效果，共享访问必须使用叫做“内存栅栏”（memory barrier）的特殊机器代码指令进行封装，从而确保共享数据读取和写入的顺序与所有正运行的线程所看到的顺序是一致的。与精心构建的缓存层次结构相比，这种序列化的代价相当昂贵。同时，坚守顺序一致性也妨碍了操作的重新排序，而这是在编译器级实现优化的重要源头。总之，这两个限制导致了运行速度的急剧下降——差不多有一个数量级。

而好消息是，这种速度损失只针对共享数据才会发生，而这种情况往往是比较少见的。在实际的项目里，大部分数据都不是共享的，因此不需要满足顺序一致性的要求。编译器会尽可

自在读书 www.zizidiary.com

能使用非共享数据来对代码进行优化，并保证其他线程不会访问到它，只能踮着脚尖眼睁睁看着共享数据。对于共享数据，有一种常见的并且值得推荐的编程风格，即将共享值复制成线程本地的工作副本，接着处理这个副本，然后将副本写回到那个共享值里。

13.13　使用 synchronized 类实现基于锁的同步

有一个历来很受欢迎的编写多线程程序的方法，即“基于锁的同步”（lock-based synchronization）。在这个准则里，共享数据的访问是被互斥量（mutex）保护的——互斥量就是同步对象，它会将某些特殊代码的执行部分序列化。这些特殊代码会临时破坏数据的一致性，或者会看到这种临时破坏，因此它们也被称为“临界区”（critical section）[①]。

对于一个基于锁的程序，其正确性是通过引入对共享数据有序、串行的访问来提供保证的。一个需要访问某块共享数据的线程必须要先获得（锁住）一个互斥量，接着操作这些数据，然后释放（解锁）这个互斥量。每次只会有一个线程可以获得给定的互斥量，从而形成序列化：当多个线程需要获得同一个的互斥量时，在某个线程“获胜”之后，其他线程则会乖乖地排队等候。排队方式，即线程优先级，是很重要的，并且可以很明显地影响到应用程序和操作系统。

可以说，多线程程序的“Hello, world!”版本就是关于银行账户的实例——一个可以从多个线程进行访问的对象，它必须开放一个安全的接口用于存款或取款。单线程的基准版本有点像下面这个样子：

```
import std.exception;

// 单线程的银行账户
class BankAccount {
   private double _balance;
   void deposit(double amount) {
      _balance += amount;
   }
   void withdraw(double amount) {
      enforce(_balance >= amount);
      _balance -= amount;
   }
   @property double balance() {
      return _balance;
   }
}
```

在可以自由线程化的世界里，+=和-=容易误导人，因为它们“看起来”很像原子操作，但实际上不是——两者都是“读取-修改-写入”操作。实际上，_balance += amount 会被编码成_balance = _balance + amount，这意味着处理器会先将_balance 和_amount 加载到它自己的操作内存里（存器或内部栈），接着将它们相加，然后将结果存回到_balance。

无保护的并发“读取-修改-写入”操作会导致不正确的行为。假如你的账户里有_balance == 100.0，接着通过存款检查调用 deposit(50)触发了一个线程。恰好就在从内存加载

① 一个潜在的让人困惑的地方，对于轻量级的保护临界区的互斥对象，Windows 使用的术语是“临界区”；而对于辅助进程间通信的更为重量级的互斥对象，使用的术语为“互斥量”。

自在读书 www.zizidiary.com

100.0 之后，这个调用会立即被另一个调用 withdraw(2.5) 的线程打断（比如，你在街角咖啡店使用借记卡买了一杯拿铁咖啡）。现假设咖啡店线程不间断地完成了整个调用，并将 _balance 更新到了 97.5。但这个事件的发生对存款线程来说是未知的，它可是已将 100 加载了到某个 CPU 寄存器里，并且仍然认为这是个正确数目。deposit(50) 调用会计算出一个新的余额 150，其后将这个数字写回到 _balance。这是一个典型的“竞态条件”（race condition）。祝喜你！你的咖啡没花钱（尽管有被警告过，错漏泛滥的书本示例还可以按你的需要进行调整，但错漏泛滥的生产代码则没办法了）。为了引入正确的同步，许多语言提供一个 Mutex 类型，它是基于锁的线程化程序，可用来保护对 _balance 的访问：

```
// 这段不是 D 语言代码
// 使用某一种语言实现的、带有显式互斥量的多线程“银行账户”
class BankAccount {
   private double _balance;
   private Mutex _guard;
   void deposit(double amount) {
      _guard.lock();
      _balance += amount;
      _guard.unlock();
   }
   void withdraw(double amount) {
      _guard.lock();
      try {
         enforce(_balance >= amount);
         _balance -= amount;
      } finally {
         _guard.unlock();
      }
   }
   @property double balance() {
      _guard.lock();
      double result = _balance;
      _guard.unlock();
      return result;
   }
}
```

针对 _balance 的所有操作现在已通过 _guard 进行了保护。表面上看起来没必要使用 _guard 来保护 _balance，因为 double 可以进行原子式读取。但这种保护是必需的，因为它们需要将自己隐藏在多层“玛雅面纱”（Maya veil）之下。总之，正因为现今有了可执行积极、优化操作的编译器和不受任何拘束的内存模型，所以所有对共享数据的访问都必须忍受有点儿奇怪的“秘密握手”（secret handshake），其中涉及内容包括写入线程、读取线程以及优化编译器。任何对共享数据的“简单读取”（bald read），都一定会将你置于一个令你痛苦不堪的世界。因此，D 语言通过设计来禁止这种简单读取，是非常了不起的。首要的一点，优化编译器在看到你没有对同步采取任何措施时，便会有权对 _balance 进行优化，采用的方法是将它放置在一个处理器寄存器里。其次，在所有示例里（最简单的除外），编译器和 CPU 有权随意地将对共享数据的简单且未做修饰的访问重新排序，因为它们认为自己可以处理线程本地的数据。这是为什么呢？因为这是最常见的情况，并且会生成最快的代码。另外，为什么要让“普通百姓”受到伤害，而不是让少数人和道德高尚的人受点苦呢？这只是让现在的多线程藐视直觉以及让精通经典多线程的程序员产生混淆的一种方式。总之，这个 balance 特性必须是同步的，这样才能保证“秘密

自在读书 www.zizidiary.com

握手”正常进行。

在出现异常和提前返回时，为保证 Mutex 可以正确地解锁，那些具有作用域对象生命周期和析构函数的语言定义了一种辅助类型 Lock，它可用来获得构造函数里的锁，并在析构函数里释放它。随之出现的惯用法叫做“作用域加锁”（scoped locking）[50]，而它在 BankAccount 里的应用有点像下面这个样子：

```
// 互锁“银行账户”的 C++版本，它使用了作用域加锁来实现
class BankAccount {
private:
   double _balance;
   Mutex _guard;
public:
   void deposit(double amount) {
      auto lock = Lock(_guard);
      _balance += amount;
   }
   void withdraw(double amount) {
      auto lock = Lock(_guard);
      enforce(_balance >= amount);
      _balance -= amount;
   }
   double balance() {
      auto lock = Lock(_guard);
      return _balance;
   }
}
```

Lock 简化了代码，并通过自动化的“锁定和解锁”配对操作增强了代码的正确性。Java、C#和其他语言通过一些方法把问题大大简化，这些方法包括将_guard 嵌入成一个隐藏的成员，以及将锁定逻辑提升至方法签名。在 Java 里，这种示例有点像下面这个样子：

```
// 互锁“银行账户”的 Java 版本，它使用了带 synchronized 声明的
//     自动作用域加锁操作来实现
class BankAccount {
   private double _balance;
   public synchronized void deposit(double amount) {
      _balance += amount;
   }
   public synchronized void withdraw(double amount) {
      enforce(_balance >= amount);
      _balance -= amount;
   }
   public synchronized double balance() {
      return _balance;
   }
}
```

相应的 C#代码与之非常相似，只是 synchronized 应替换成[MethodImpl(MethodImplOptions.Synchronized)]。

没错，你刚看到了好消息：在小规模情形里，基于锁的编程容易理解，并且工作得很好。而坏消息是：在大规模情形里，难以将数据正确地配对锁住，难以选择要锁定的作用域和粒度，以及在跨多个对象时难以一致地使用锁。如果不注意后面那个问题，则会导致线程在“死锁”（deadlock）状态彼此等待。在经典多线程的美好旧时代里，这些问题让基于锁的编程变得非常困难。当代的多线程技术（具有大规模并发性、不受拘束的内存模型，以及代价昂贵的数据共

自在读书 www.zizidiary.com

享这样一些特点）已逐渐开始反对采用基于锁的编程方式[53]。不管怎么样，基于锁的同步在各种设计当中仍然很有用。

对于基于锁的同步，D 语言提供的机制比较有限。这些限制是人为的，其好处在于能提供强有力的担保。对于 BankAccount 这一特定情形，其 D 语言版本非常简单：

```
// 互锁“银行账户”的 D 语言版本，它使用了一个 synchronized 类来实现
synchronized class BankAccount {
   private double _balance;
   void deposit(double amount) {
      _balance += amount;
      }
   void withdraw(double amount) {
      enforce(_balance >= amount);
      _balance -= amount;
   }
   double balance() {
      return _balance;
   }
}
```

D 语言将 synchronized 提升了一级，让它作用到整个类。这样，D 语言的 BankAccount 便能提供更强有力的保证：即使你想犯错误，也会没有办法从后门提供对 _balance 的不同步访问。如果 D 语言允许在同一个类里混合使用 synchronized 和非 synchronized 方法，那么所有努力都将前功尽弃。事实上，在方法级进行同步的经验表明：最好是要么全部定义为 synchronized，要么全都不定义。具有双重目的类所带来的麻烦远比它们带来的好处多。

在类一级的 synchronized 属性会影响到类型为 shared(BankAccount) 的对象，并且会自动序列化对所有类方法的调用。还有，对 synchronized 类的保护性检查会变得更为严格。回想一下 11.1 节的内容，一般的保护检查通常都会允许模块内的所有代码对非公共成员进行访问。对于 synchronized 类可不是这样的，它遵从下面这些规则。

- 根本就不允许 public 数据。
- 对 protected 成员的访问被限制成对类及其后代的方法进行调用。
- 访问 private 成员的访问被限制成对类方法进行调用。

13.14　synchronized 类里的字段类型化

用于 shared 对象的传递性规则规定 shared 类对象会将 shared 修饰符传播到它的所有字段。很显然，synchronized 对于那张表会附加一些新的规则要求。在 synchronized 类的方法内部，那张表反映出来的字段检查有点宽松。为了提供强有力的保证，synchronized 需要以一种稍显奇特的方式对字段的语义检查实加影响，即追踪 synchronized 所特有的语义。

同步方法针对竞争的保护是临时的和局部的。其临时性体现在：只要方法返回，那些字段将不再因竞争而需要保护。而局部性体现在：synchronized 会对直接嵌入该对象内部的数据给以保护，而对该对象间接引用的数据（即通过类引用、指针或数组）则不提供保护。现在依次看看每一种情况。

自在读书 www.zizidiary.com

13.14.1　临时保护==无脱离

也许不是很直观，synchronized 的临时性质涉及这样的规则，即字段的地址都不可以脱离 synchronized 类/对象/方法。如果出现这种情况，那么代码的其他部分则可能会访问到某些超出由方法级同步所赋予的临时保护之外的数据。

编译器会拒绝所有这样的操作，即它会返回指向方法里某个字段的引用或指针，也会拒绝通过引用或指针将一个字段传递到某个函数。为了说明这条规则为什么是很明智的，请看下面的示例：

```
double * nyukNyuk;                                  // 注意：不是共享的

void sneaky(ref double r) { nyukNyuk = &r; }

synchronized class BankAccount {
   private double _balance;
   void fun() {
      nyukNyuk = &_balance;                         // 错误！（理所当然的）
      sneaky(_balance);                             // 错误！（理所当然的）
   }
}
```

fun 的第一行试着想要获得_balance 的地址，然后将它分配给一个全局变量。如果此操作成功，那么类型系统的保证就会失效——从此以后，这个程序便可以共享访问到一个非共享的值。这个赋值操作无法通过类型检查。第二个操作有点奇妙，它试图想要通过调用一个接受 ref 类型参数的函数来实现别名化操作，这也会失败。实际上，通过 ref 方式传递一个值也需要在调用之前获取地址，而获取地址是被禁止的，因此这个调用会失败。

13.14.2　局部保护==尾部共享

synchronized 所提供的保护在某种意义上也是局部的，它没有必要保护此对象的直属字段以外的数据。只要涉及间接性，只能有一个线程可以进行数据访问的保证就会失效。如果将数据当做是由“头部”（部分存在于 BankAccount 对象所占用的物理内存）和可能存在的“尾部”（间接访问的内存）构成，那么 synchronized 类能够保护数据“头部”，而“尾部”仍然为 shared。鉴于这一事实，synchronized 类的字段在方法内的类型化过程如下。

- 所有的数字类型都不为 shared（它们没有“尾部”），因此可以对它们进行正常的处理。
- 声明为 T[]类型的数组字段可以接收类型为 shared(T)[]的数据。也就是说，“头部”（分割限制）不为 shared，而“尾部”（数组内容）会保持为 shared。
- 声明为 T*类型的指针字段可以接收类型为 shared(T)*的数据。也就是说，“头部”（指针本身）不为 shared，而“尾部”（指向的数据）会保持为 shared。
- 声明为 T 类型的类字段可以接收类型为 shared(T)的数据。类自动就是通过引用访问的，所以它们都是“尾部”。

自在读书 www.zizidiary.com

这些规则是建立在前一节所描述的“无脱离”规则之上的。一种直接的后果是，这些影响对象的直属字段的操作可以在这种方法内部随意地重新排序和优化，仿佛共享暂时对它们失去了作用——而这正是 synchronized 所做的事情。

也存在这样的情况：一个对象完全拥有另一个对象。例如，BankAccount 将它过去所有的事务都保存在一个 double 类型的列表里：

```
// 不进行同步，并且通常情况下也不考虑多线程
class List(T) {
    ...
    void append(T value) {
        ...
    }
}

// 保留一连串的事务
synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;
    void deposit(double amount) {
        _balance += amount;
        _transactions.append(amount);
    }
    void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
        _transactions.append(-amount);
    }
    double balance() {
        return _balance;
    }
}
```

List 类并未设计成支持跨线程共享，因此它没有使用任何同步机制，不过它实际上也从未被共享！它所有的用法相对于 BankAccount 对象完全都是私有的，并且完全被保护在 synchronized 方法内部。假如 List 没有干什么蠢事，如将某个内部指针保存为一个全局变量，那么这段代码应该是完全没有问题的。

很不幸，它还是有问题。上面的代码在 D 语言里是无法工作的，因为 append 对 shared(List!double)对象来说是可调用的。编译器之所以会拒绝，其中一个很突出的原因就是信用系统与编译器不相容。List 可能算是一个表现不错的类，但编译器仍然需要更确凿一点的证据才能知道不会出现共享数据被偷偷摸摸别名化的情况。理论上，编译器可以继续检查 List 的类定义；但紧接着，List 可能会使用某些在其他模块里找到的组件，并且在你能够说出“进程间分析”之前，事情已失去控制。

“进程间分析”（interprocedural analysis）是一种编译器和分析器用来证明关于程序事实的技术，它通过一次查看多个函数的方法实现。这种分析通常很慢，对于程序大小的可伸缩性很差，而且与单独编译向来是“宿敌”。尽管有系统使用了进程间分析，但当今大部分语言（包括 D 语言）在进行类型检查时都不会要求它。

对这个从属子对象问题的另一种解决方案是为这种描述所属关系的对象再添加一个新的修饰符，例如，“BankAccount 拥有其自己的_transactions 成员，因此它的互斥量也会对

自在读书 www.zizidiary.com

_transactions 上的操作进行序列化。”有了恰当的说明，编译器据此可以确认 _transactions 完全封装在 BankAccount 内部，因此它可以安全地使用，而不用担心过度共享的问题。也有系统和语言提议采用这种方法[25，2，11，6]，但它们暂时还未成为主流。这种所属关系系统在语言和编译器里会引起很大的混乱。由于基于锁的同步完全受到排斥，因此 D 语言对它也是避而远之，没再去支持这种已处于“病态”的编程技术。这个问题以后还可能会被再次提起（所属关系系统已经被提议用于 D 语言[42]），但目前某些基于锁的设计肯定会脱离类型系统的范畴，接下来会讨论到这一点。

13.14.3　强制同质的互斥量

D 语言动态支持类型系统无法静态保证的功能：针对“锁定”的所属关系。下面的全局基础函数是可访问的：

```
// 在 object.d 内部
setSameMutex(shared Object ownee, shared Object owner);
```

类对象 obj 可以调用 obj.setSameMutex(owner) 有效地将其关联的同步对象抛开，然后开始使用与 owner 一样的同步对象。这样，你便可以确信在锁定 owner 时实际上也会将 obj 锁住。我们来看一看 BankAccount 和 List 是如何协调工作的。

```
// 线程相关
synchronized class List(T) {
   ...
   void append(T value) {
      ...
   }
}

// 保留一连串的事务
synchronized class BankAccount {
   private double _balance;
private List!double _transactions;

   this() {
      // 此账户拥有该列表
      setSameMutex(_transactions, this);
   }
   ...
}
```

此方案的工作方式要求 List（从属对象）为 synchronized。紧接着对 _transactions 的操作会基于每一个普通的规则将 _transactions 锁住，但事实上它们会继续直接获得 BankAccount 对象的互斥量。编译器很喜欢这种方式，因为它会认为每个对象都是被单独锁住的。同时，这个程序也会很开心，因为事实上只有一个互斥量控制着 BankAccount 以及 List 子对象。获取 _transactions 的互斥量事现上就是获取已经锁定的 this 互斥量。庆幸的是，这种递归获取一个已被占有的、无竞争的锁的操作的代价相对低廉，因此这段代码是正确的，而且没有密集的锁定操作。

自在读书 www.zizidiary.com

13.14.4 不可思议：shared 可被强制转换掉

继续上面的示例，如果非常肯定_transactions 列表对于 BankAccount 对象完全是私有的，那么可以将 shared 强制转换掉，并可以像下面这样在不考虑任何线程的情况下使用它：

```
// 不进行同步，并且通常情况下也不考虑多线程
class List(T) {
    ...
    void append(T value) {
        ...
    }
}

synchronized class BankAccount {
    private double _balance;
    private List!double _transactions;
    void deposit(double amount) {
        _balance += amount;
        (cast(List!double) _transactions).append(amount);
    }
    void withdraw(double amount) {
        enforce(_balance >= amount);
        _balance -= amount;
        (cast(List!double) _transactions).append(-amount);
    }
    double balance() {
        return _balance;
    }
}
```

现在，这段代码能够正常编译和运行。唯一需要注意的是，在这个程序里，对于基于锁的原则，其正确性现在需要由你来确保，而不是依靠语言的类型系统。因此，如果你对 D 语言还不够了解的话，请使用默认的共享方法。你仍然可以享受的好处是强制类型转换是局部的，可以进行查找和仔细的复查。

13.15 死锁与 synchronized 语句

如果那个银行账户示例是线程化程序的“Hello, world!”，那么银行账户转账示例则一定正好对应于对线程死锁的介绍（或许更糟糕的情况）。这个示例有点像这个样子：假设你有两个 BankAccount 对象，如 checking 和 savings，所面临的挑战是要定义一个可以将部分资金从一个账户转移到另一个账户的原子转账器。

有种非常简单的方法像下面这个样子：

```
// 转账器版本 1：非原子操作
void transfer(shared BankAccount source, shared BankAccount target,
        double amount) {
    source.withdraw(amount);
    target.deposit(amount);
}
```

自在读书 www.zizidiary.com

不过，这个版本不是原子操作。在那两个调用之间有一小段时间，两个账户的钱都不见了。如果恰在那时有一个线程执行了 inspectForAuditing 函数，那么事情可能麻烦了。

为了让这个转账器属于原子操作，需要在 transfer 的开始处，从那两个对象的方法里获得它们的隐藏互斥量。借助于 synchronized 语句，可以达到这种效果：

```
// 转账器版本 2：有问题
void transfer(shared BankAccount source, shared BankAccount target,
      double amount) {
   synchronized (source) {
      synchronized (target) {
         source.withdraw(amount);
         target.deposit(amount);
      }
   }
}
```

synchronized 语句在语句体的整个执行过程中获得了对象的隐藏互斥量。所有针对该对象的方法调用都从一个已经获得的锁里受益。

第二版转账器的问题在于容易出现死锁：如果两个线程都想要执行在相同账户之间的转账，只是方向相反，那么这些线程可能会永远阻塞。一个想要将资金从 checking 转移到 savings 的线程会将 checking 锁定，而恰在这时另一个正想要将资金从 savings 转移到 checking 的线程会设法将 savings 锁定。这时，每个线程都拥有锁，而每一个线程都需要另一个线程的锁。它们绝不会得到想要的结果。

要真正解决这个问题，需要使用带两个参数的 synchronized：

```
// 转账器版本 3：正确
void transfer(shared BankAccount source, shared BankAccount target,
      double amount) {
   synchronized (source, target) {
      source.withdraw(amount);
      target.deposit(amount);
   }
}
```

在同一条 synchronized 语句里，多个对象之间的同步不同于它们单个地依次同步。在所有线程里，生成的代码总是以相同的顺序获得各个互斥量，这种顺序与指定这些对象的语法顺序无关。这样，避免了死锁。

在编译器的参考实现里，实际的顺序是对象地址的递增顺序。所有的全局排序都可以工作。

多参数的 synchronized 很有用，不过很不幸，它不是什么灵丹妙药。一般的死锁不会发生在局部——可以在一个函数里获得一个互斥锁量，然后在另一个不同的函数里获得另一个互斥锁量等，一直到死锁循环结束。但是，具有多个参数的 synchronized 提升了对该问题的认识，并使用模块化的互斥量获取方式形成了正确的代码。

13.16　使用 shared 类进行无锁编码

基于锁的同步理论起始于 20 世纪 60 年代。早在 1972 年[23]，研究人员便开始慢慢地想办法

自在读书 www.zizidiary.com

在多线程程序里尽可能避免使用慢的、笨手笨脚的互斥量。例如，有些类型是自动可赋值的，因此人们认为没有必要再使用互斥量假装去守护这种赋值。此外，有些处理器提供了更为高级的轻量级互锁指令，如“原子增量”（atomic increment）或“测试并设置”（test-and-set）。在 30 年后的 1990 年，燃起了一道希望之火，多个原子式读写寄存器的巧妙结合有助于防止锁的肆虐。就这一点而言，一项具有开创性的工作在一个行业已走到了最后一步，而在另一个行业却是一个全新的开始。

Herlihy 于 1991 年发表的论文“Wait-free synchronization”[31]标志着在并发编程方面有一个突破性的发展。在那之前，硬件和软件开发人员同样都不清楚什么样的同步原语是最有效的。例如，对整数来说，一个具有原子式读和写操作能力的处理器直觉感受可能还没有原子式的“+=”强大。表面上看起来，支持原子式操作的“*=”甚至会更好些。一般情况下，可以随意使用的原子式原语肯定是多多益善。

Herlihy 极力鼓吹这个理论，还特意展示了一些看似强大的同步原语，如“测试并设置”（test-and-set）、“获取并添加”（fetch-and-add），以及一个全局共享 FIFO 队列，而这些几乎都是没用的。这些“不可能有的结果”很容易证明，无需多说即可消除人们心里的错觉：这个机制能够提供具有魔法性的并发性药水。幸运的是，Herlihy 也已证明了“普遍性结果”——某些同步原语理论上可以同步数量不限的并发线程。不过值得注意的是，“优秀的”原语实现起来并不比“糟糕的”原语困难多少，并且肉眼看来也可能没那么强大。有一个很有用的、被称为“比较并交换”（compare-and-swap，简写为 cas）的同步原语，现在已逐渐流行起来；而且，现今几乎所有的处理器都实现了这一原语。“比较并交换”原语具有以下语义：

```
// 这个函数会以原子方式执行
bool cas(T)(shared(T) * here, shared(T) ifThis, shared(T) writeThis) {
    if (*here == ifThis) {
        *here = writeThis;
        return true;
    }
    return false;
}
```

在普通的语言里，cas 将一个内存位置与一个给定值进行原子式地比较。如果这个位置等于那个值，那么它便存储一个新的值；否则，什么也不做。此操作的结果是想测试出这个存储操作是否会发生。整个 cas 操作都是原子的，并且一定是以原语形式提供。T 可能有的集合仅限于主机的原生字大小所形成的整数（即 32 或 64 位）。越来越多的机器提供“双字的比较并交换”（double-word compare-and-swap）原语，有时也被称作 cas2。该操作可以以原子方式在 32 位机器上处理 64 位的数据，以及在 64 位机器上处理 128 位的数据。鉴于越来越多的当代机器都支持 cas2 这一现状，D 语言使用了同一名称（即 cas）来重载原有函数，从而提供了对“双字的比较并交换”这一原语的支持。因此在 D 里，你可以对多种类型的值执行“比较并交换”操作，这些类型包括 int、long、float、double、所有数组、所有指针和所有类引用。

13.16.1　shared 类

有了 Herlihy 的普遍性证明，许多数据结构和算法围绕着新生的“基于 cas 的编程”开始脱

自在读书 www.zizidiary.com

胎换骨。现在，如果一个基于 cas 的实现理论上可以解决任何同步问题，那么不会再有人说它是简单的了。定义基于 cas 的数据结构和算法，特别是要证明它们都可以正确地工作，是一件极为艰难的事。庆幸的是，一旦定义和封装好了这种实体，它便可以被反复使用，并带来无穷的好处[57]。

为利用基于 cas 的无锁操作所带来的好处，可以在类或结构的定义里使用 shared 属性：

```
shared struct LockFreeStruct {
   ...
}

shared class LockFreeClass {
   ...
}
```

平常的传递性规则在这里也适用：shared 会传播到结构和类的字段，而对方法则未提供特别的保护。所有你能得到的好处包括原子式赋值、cas 调用、编译器和机器不会对操作进行任何重新排序的保证，以及你“肆无忌惮”的自信心。不过请注意：如果编码是行走，消息传递是慢跑，那么无锁编程则不亚于参加奥运会比赛[①]。

13.16.2　几种无锁结构

作为热身运动，我们来实现一个无锁的栈类型。其基本思想比较简单，即栈就是一个单链表，插入和删除都在前端进行：

```
shared struct Stack(T) {
   private shared struct Node {
      T _payload;
      Node * _next;
   }
   private Node * _root;

   void push(T value) {
      auto n = new Node(value);
      shared(Node)* oldRoot;
      do {
        oldRoot = _root;
        n._next = oldRoot;
      } while (!cas(&_root, oldRoot, n));
   }

   shared(T)* pop() {
      typeof(return) result;
      shared(Node)* oldRoot;
      do {
         oldRoot = _root;
         if (!oldRoot) return null;
         result = & oldRoot._payload;
      } while (!cas(&_root, oldRoot, oldRoot._next));
      return result;
   }
}
```

① 作者言下之意是说，无锁编程的工作量比较大。——译者注

自在读书 www.zizidiary.com

Stack 是一个 shared 结构，由此带来的直接后果是其内部几乎所有内容都为 shared。内部类型 Node 是经典的“净负载和指针”（payload-and-pointer）结构，而 Stack 自身则存储了这个列表的根节点。

两个原语里的 do/while 循环看起来可能有点儿古怪，但它们都很常见。慢慢地，它们会让每一位基于 cas 编程的专家头痛不已。push 的工作方式为：先创建一个新的 Node，用于存储新的值；然后，在一个循环里，_root 被赋与了指向这个新节点的指针，但同时要求其他线程不会更改它！另一个线程也在执行栈操作，这是有可能的。因此，push 需要确保在新的节点被填写时，oldRoot 里设定的根节点未发生变化。

pop 方法不是按值返回，而是通过指针返回的。这是因为 pop 可能会发现这个队列是空的，而这种情形并不是一个异常条件（因为在一个单线程栈里它也可能存在）。对于一个共享栈来说，检查元素、删除元素以及返回元素是一组相关的操作。除了返回操作，pop 与 push 的实现很相似：_root 被很小心地替换掉，这样在获取净负载时其他线程才不会更改它。在循环末尾，提取出的值脱离了这个栈，并且可以安全地返回给它的调用者。

如果 Stack 看上去没那么复杂，那么接下来再看一个内容更为丰富的单链接口。要知道，大部分的基础设施都已在 Stack 里建好。

遗憾的是，对列表来说，事情肯定会变得更加困难。到底有多困难呢？答案是困难到了极点。最根本的问题在于在列表的任意位置插入和删除节点。假设我们有一个 int 列表，它包含有一个负载为 5 的节点，后面紧跟着一个负载为 10 的节点，而我们想要删除节点 5。这时没有问题——只需要施展 cas 的魔法，将_root 指向节点 10 即可。问题在于，如果在这个时候又有另一个线程恰好在节点 5 的后面插入了一个新的节点，那么这个新节点将会彻底地被丢掉；但_root 对此却一无所知。

理论上有好几种解决方案，但它们当中没有一个是简单的。下面所描述的实现是 Harris[30] 在题为“A pragmatic implementation of non-blocking linked-lists”的论文中首先提出的，很有独创性，因为它采用的方法是设置那个_next 指针的未曾使用的最低有效位。其主要思想是：首先将那个指针标记为“逻辑上被删除的”，所用方法是将那个位设置为零；接下来的第二步是将这个节点完全切除：

```
shared struct SharedList(T) {
   shared struct Node {
      private T _payload;
      private Node * _next;

      @property shared(Node)* next() {
         return clearlsb(_next);
      }

      bool removeAfter() {
         shared(Node)* thisNext, afterNext;
         // 第一步：为要删除的节点设置_next 的最低有效位
         do {
            thisNext = next;
            if (!thisNext) return false;
            afterNext = thisNext.next;
         } while (!cas(&thisNext._next, afterNext, setlsb(afterNext)));
```

自在读书 www.zizidiary.com

```
            // 第二步：切除要删除的节点
            if (!cas(&_next, thisNext, afterNext)) {
                afterNext = thisNext._next;
                while (!haslsb(afterNext)) {
                    thisNext._next = thisNext._next.next;
                }
                _next = afterNext;
            }
        }

        void insertAfter(T value) {
            auto newNode = new Node(value);
            for (;;) {
                // 试着查找一个插入点
                auto n = _next;
                while (n && haslsb(n)) {
                    n = n._next;
                }
                // 找到一个可能的插入点，试着插入
                auto afterN = n._next;
                newNode._next = afterN;
                if (cas(&n._next, afterN, newNode)) {
                    break;
                }
            }
        }
    }

    private Node * _root;

    void pushFront(T value) {
        ... // 等同于 Stack.push
    }

    shared(T)* popFront() {
        ... // 等同于 Stack.pop
    }
}
```

这个实现有点复杂，但如果你能多记住几个不变量的话，也还可以理解。首先，逻辑上被删除的多个节点（即需要对这些 Node 对象的字段_next 进行设置，设置的地方是最低有效位）仍需要留存一下，这是很正常的。其次，一个节点绝不能插入到一个已逻辑上被删除的节点后面。这样，即使有些节点可能在任何时间出现和消失，这个列表也会保持连贯。

clearlsb、setlsb 和 haslsb 的实现则有些原始。例如：

```
T* setlsb(T)(T* p) {
    return cast(T*) (cast(size_t) p | 1);
}
```

13.17　小结

setlsb 的实现尽管不太美观，但也还适合作为一章的结尾。本章从简单、漂亮的消息传递开始，逐渐演变成了关于共享的“地狱”。

D 语言提供了丰富的线程工具。对于当代机器上的大部分应用程序，首选机制是围绕着消

自在读书 www.zizidiary.com

息传递来定义协议。不可变的共享也能提供很大的帮助。强烈建议使用消息传递来定义健壮的、可伸缩性好的并发应用程序。

如果需要基于互斥的同步操作，那么可以借助于 synchronized 类。请注意，与其他语言相比，对于基于锁的编程，D 语言所提供的支持有所限制，并且其中有着很充分的理由。

如果需要简单的数据共享，可以使用 shared 值。D 语言会保证针对这些值的操作会以这些值在代码里指定的顺序进行，并且不会导致可见性悖论和底层竞争。

最后，如果像蹦极、驯服鳄鱼或者走火碳这样的活动对你来说有些无聊透顶的话，那么你可能会很高兴看到有无锁编程方式的存在。在 D 语言里，你可以通过使用 shared 结构和类来达到这一目的。

自在读书
www.zizidiary.com

参考文献

[1] ALAGIĆ, S., AND ROYER, M. Genericity in Java: Persistent and database systems implications. *The VLDB Journal* 17, 4 (2008), 847–878.

[2] ALDRICH, J., KOSTADINOV, V., ANDCHAMBERS, C. Alias annotations for program understanding. In *OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, 2002), ACM Press, pp. 311–330.

[3] ALEXANDRESCU, A. On Iteration. *InformIT* (November 2009). `http://erdani.com/publications/on-iteration.html`.

[4] AMSTERDAM, J. Java's new considered harmful. *Dr. Dobb's Journal* (April 2002). `http://www.ddj.com/java/184405016`.

[5] ARMSTRONG, J. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2007.

[6] BACON, D. F., STROM, R.E., ANDTARAFDAR, A. Guava: A dialect of Java without data races. In *OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications* (NewYork, NY, 2000), ACM Press, pp. 382–400.

[7] BENOIT, F., REIMER, J., ANDCARLBORG, J. The D Widget Toolkit (DWT). `http://www.dsource.org/projects/dwt`.

[8] BLOCH,J.*Effective Java Programming Language Guide*. Sun Microsystems, Inc.,Mountain View, CA, 2001.

[9] BLOCH, J. *Effective Java, Second Edition*. Prentice Hall PTR, Upper Saddle River, NJ, 2008.

[10] BÖHM, C., AND JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM 9*, 5 (1966), 366-371.

[11] BOYAPATI, C., LEE, R., ANDRINARD, M. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, 2002), ACM Press, pp. 211–230.

[12] BRIGHT, W. The D assembler. `http://digitalmars.com/d/1.0/ iasm.html`.

[13] BROOKS, JR., F. P. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley, Reading, MA, August 1995.

[14] CABANA, B., ALAGI´ C, S., ANDFAULKNER, J. Parametric polymorphism for Java: Is there anyhope in sight? *SIGPLAN Notices 39*, 12 (2004), 22–31.

[15] CARDELLI, L. Type Systems. In *The Computer Science and Engineering Handbook*, A. B.

自在读书 www.zizidiary.com

Tucker, Ed. CRC Press, Boca Raton, FL, 1997, ch. 103, pp. 2208–2236.

[16] CHANG, R. "Near Speed-of-Light On-Chip Electrical Interconnects". PhD thesis, Stanford University, 2003.

[17] COHEN, T. Java Q&A: How do I correctly implement the equals() method? *Dr. Dobb's Journal* (May 2002). `http://www.ddj.com/ java/184405053.`

[18] DIGITAL MARS. dmd—FreeBSD D Compiler. `http://digitalmars.com/ d/2.0/dmd-freebsd.html`, 2009.

[19] DIGITAL MARS. dmd—Linux D Compiler. `http://digitalmars.com/d/2.0/dmd-linux.html`, 2009.

[20] DIGITAL MARS. dmd—OSX D Compiler. `http://digitalmars.com/d/2.0/dmd-osx.html`, 2009.

[21] DIGITAL MARS. dmd—Windows D Compiler. `http://digitalmars.com/d/2.0/dmd-windows.html`, 2009.

[22] DREPPER, U. What every programmer should know about memory. *Eklektix, Inc.* (October 2007).

[23] EASTON, W. B. Process synchronization without long-term interlock. *ACM SIGOPS Operating Systems Review 6*, 1/2 (1972), 95–100.

[24] FINDLER, R. B., ANDFELLEISEN, M. Contracts for higher-order functions. *ACM SIGPLAN Notices 37*, 9 (2002), 48–59.

[25] FLANAGAN, C., ANDABADI, M. Object types against races. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*(London, UK, 1999), Springer-Verlag, pp. 288–303.

[26] FRIEDL, J. *Mastering Regular Expressions. O*'Reilly Media, Inc., Sebastopol, CA, 2006.

[27] GAMMA, E., HELM, R., JOHNSON, R., ANDVLISSIDES, J.*Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[28] GOVE, D. *Solaris Application Programming*. Prentice Hall PTR, Upper Saddle River, NJ, 2008.

[29] GROPP, W., LUSK, E., ANDSKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambdridge, MA, 1999.

[30] HARRIS, T. L. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science 2180* (2001), 300–314.

[31] HERLIHY, M. Wait-free synchronization. *TOPLAS: ACM Transactions on Programming Languages and Systems* 13, 1 (1991), 124–149.

[32] HOFFMAN, D. M., AND WEISS, D. M., Eds. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, Reading, MA, 2001.

[33] ISO. The ANSI C standard (C99). Tech. Rep. WG14 N1124, ISO/IEC, 1999.

自在读书 www.zizidiary.com

[34]KERNIGHAN, B.W., ANDRITCHIE, D. M. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.

[35] KNUTH, D. E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1997.

[36] KORPELA, J. K. *Unicode Explained*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[37] LEE, E. A. The problem with threads. *Computer 39*, 5 (2006), 33–42.

[38] LISKOV, B. Keynote address—Data Abstraction and Hierarchy. In *OOPSLA: Object-Oriented Programming, Systems, Languages, and Applications*（New York, NY, 1987）, pp. 17–34.

[39] MARTIN, R. C. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, October 2002.

[40] MEYER, B. *Object-Oriented Software Construction*. Prentice Hall, Inc., Upper Saddle River, NJ, 1988.

[41] MEYERS, S. How non-member functions improve encapsulation. *C++ Users Journal* 18, 2 (2000), 44–52.

[42] MILEWSKI, B. Race-free multithreading: Ownership. Blog post at `http://bartoszmilewski.wordpress.com/2009/06/02/race- free-multithreadingownership/`.

[43] ODERSKY, M., ANDWADLER, P. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Paris, France, 1997), ACM, pp. 146–159.

[44] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[45] PARNAS, D. L. A technique for software module specification with examples. *Commun. ACM* 15, 5 (1972), 330–336.

[46] PIERCE, B. C. *Types and Programming Languages*. MITPress, Cambridge, MA, 2002.

[47] PIKE, R. UTF-8history. `http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history. txt`, 2003.

[48] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., ANDFLANNERY, B. P. *Numerical Recipes: The art of scientific computing*. Cambridge University Press, New York, NY, 2007.

[49] RADENSKI, A., FURLONG, J., ANDZANEV, V. The Java 5 generics compromise orthogonality to keep compatibility. *J. Syst. Softw. 81*, 11 (2008), 2069–2078.

[50] SCHMIDT, D. C. Strategized locking, thread-safe interface, and scoped locking.C++ Report 11, 9 (1999).

[51] STEPANOV, A., ANDLEE, M. The Standard Template Library. Tech. rep., WG21 X3J16/94-0095, 1994.

[52] SUTTER, H. Virtuality. *C/C++ Users Journal* (September 2001). `http://www.gotw.ca/publications/mill18.htm`.

自在读书 www.zizidiary.com

[53] SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30*, 3 (2005), 202–210.

[54] SUTTER, H. Use threads correctly = isolation + asynchronous messages. *Sutter's Mill* (March 2009). `http://herbsutter.com/ 2009/03/16/`.

[55] SUTTER, H., AND ALEXANDRESCU, A. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, Reading, MA, 2004.

[56] THE UNICODE CONSORTIUM. *The Unicode Standard, Version 5.0*. Addison-Wesley, Reading, MA, 2006.

[57] VALOIS, J. D. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (1995), ACM NewYork, NY, pp. 214–222.

[58] VONRONNE, J., GAMPE, A., NIEDZIELSKI, D., ANDPSARRIS, K. Safe bounds check annotations. *Concurrency and Computation: Practice and Experience 21*, 1 （2009）.

[59] WADLER, P. Proofs are programs: 19th century logic and 21st century computing. *Dr. Dobb's Journal* (December 2000).

[60] WEGNER, P. A technique for counting ones in a binary computer. *Communications of the ACM 3*, 5 (1960), 322.

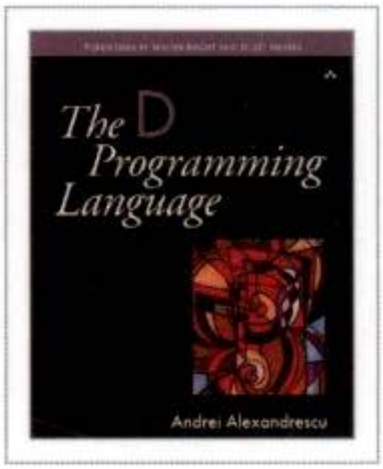
[61] XU, D.N., PEYTONJONES, S., ANDCLAESSEN, K. Static contract checking for Haskell. *SIGPLAN Not. 44*, 1 (2009), 41–52.

北航 C1645675

自在读书 www.zizidiary.com

D程序设计语言

The D Programming Language



D语言是一门用于帮助程序员解决现代软件开发中所遇难题的编程语言。为实现这一目标，它提供了接口相互连接的模块、紧密集成的多种编程范型、由语言实施的线程隔离、模块化的类型安全和高效的内存模型等内容。

本书是一本权威的、全面介绍D语言的书籍。它反映了作者别具一格的创作风格，其谈话式的写作方式，让人感觉很放松，但却从未模糊焦点或失去准确性。它涵盖了语言的各个方面（如表达式、语句、类型、函数、契约和模块），但它并没有简单罗列这些特性。

在本书中，你还会发现以下内容：

- 通过惯用示例，对所有语言特性都进行了深入的解释；
- 如何将语言特性进行组合以支持重要的编程范型；
- 对每一个主要特性的基本原理都进行了说明，并对这些特性的最佳用途提出了建议；
- 对跨领域的问题（如错误处理、契约编程和并发性）进行了讨论；
- 包含大量的表格、图片和速查表，它们为使用D语言解决日常问题提供了快速参考。

本书为工作在一线的程序员而作，它不仅介绍了D语言，还提供了大量优秀的实践与惯用法，以帮助你使用D语言进行编码和实现通用性编码。

Andrei Alexandrescu博士　获奖图书*Modern C++ Design*与*C++ Coding Standards*（与Herb Sutter合著）的作者。Andrei的工作让他名声大噪，也让他成为了编程语言和编程方法革新的领军人物。自2006年以来，他与Walter Bright（D语言的缔造者与实现者）一直保持着密切合作，参与了D语言及其标准库的设计与实现。

张雪平　2006年毕业于西南石油大学，模式识别与智能系统专业，硕士研究生。拥有多年高校计算机相关课程的教学经验，热衷于计算机应用软件的开发与研究。现从事石油行业相关软件的研制和开发工作。

谭丽娜　英语语言专业，本科毕业后一直从事一线英语教学工作。曾参与多个英语语言相关课题，其中包括国家级课题一个，省级4个，发表相关文章20余篇。

PEARSON
www.pearson.com

人民邮电出版社-信息技术分社
http://weibo.com/ptpitbooks

分类建议：计算机／程序设计
人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

ISBN 978-7-115-31419-2

ISBN 978-7-115-31419-2
定价：69.00 元